КЛАССИКА COMPUTER SCIENCE

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ в C++

4-Е ИЗДАНИЕ



Р. ЛАФОРЕ

SAMS





Р. ЛАФОРЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ в C++

4-Е ИЗДАНИЕ



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2004

Краткое содержание

Содержание

Глава 5. Функции . . . 168

Глава 6. Объекты и классы . . . 217

Глава 8. Перегрузка операций. 312

Глава 9. Наследование 361

Глава 10. Указатели 411

Эта книга посвящается GGL и ее неукротимому духу

Предисловие

Наиболее существенные изменения в 4-м издании касаются более раннего ознакомления читателя с языком UML, дополнительных сведений о межфайловом взаимодействии в главе 13 и обновления материала, относящегося к разработке программного обеспечения, в главе 16.

UML-диаграммы являются удобным иллюстративным средством. Поэтому, чтобы иметь возможность активно использовать эти диаграммы в книге, мы сочли целесообразным как можно раньше ознакомить читателя с основами языка UML. Раздел книги, посвященный межфайловому взаимодействию, объединяет большой объем информации, ранее содержавшейся в разных разделах. Подходы к созданию промышленных программ претерпели изменения со времени последнего издания, поэтому соответствующие модификации, отражающие новые тенденции, были внесены и в эту книгу.

Со времени выхода в свет последнего издания язык C++ практически не изменился. Тем не менее кроме вышеупомянутых исправлений и дополнений мы постарались сделать стиль изложения материала более ясным, а также исправить неточности и опечатки, найденные в тексте предыдущего издания.

Введение

Основная задача этой книги — научить вас создавать программы на языке C++, однако, разумеется, только этим мы не ограничимся. В последние несколько лет в сфере программного обеспечения произошел ряд значительных изменений, о которых мы расскажем чуть позже. Поэтому еще одна задача книги — изложить концепции языка C++ в контексте развития программного обеспечения.

Новые концепции программирования

Двадцать и более лет назад программисты реализовывали свои проекты путем непосредственного написания кода. С возрастанием размера и сложности проектов становилось все яснее, что такой подход неудачен. Проблема заключалась в непропорциональном возрастании сложности процесса создания самих программ.

Пожалуй, большие программы можно без преувеличения назвать самым сложным творением человека. Из-за своей сложности такие программы нередко содержат ошибки. Ошибки в программном обеспечении потенциально могут стать причиной материального ущерба, а иногда и угрожать жизни людей (например, при управлении авиаполетами). В результате борьбы с проблемой сложности программного кода были выработаны три новые концепции программирования:

- объектно-ориентированное программирование (ООП);
- унифицированный язык моделирования (UML);
- специализированные средства разработки программного обеспечения.

В данной книге, наряду с изучением C++, уделяется внимание и упомянутым концепциям, что позволяет не просто выучить язык программирования, но и получить представление об эффективных способах разработки программного обеспечения.

Объектно-ориентированное программирование

Почему объектно-ориентированный подход к программированию стал приоритетным при разработке большинства программных проектов? ООП предлагает новый мощный способ решения проблемы сложности программ. Вместо того

чтобы рассматривать программу как набор последовательно выполняемых инструкций, в ООП программа представляется в виде совокупности объектов, обладающих сходными свойствами и набором действий, которые можно с ними производить. Возможно, все то, о чем мы до сих пор говорили, будет казаться вам непонятным, пока вы не изучите соответствующий раздел программирования более подробно. Но со временем вы не раз сможете убедиться в том, что применение объектно-ориентированного подхода делает программы понятнее, надежнее и проще в использовании.

Унифицированный язык моделирования

Унифицированный язык моделирования (UML) — это графический язык, включающий в себя множество различных диаграмм, помогающих специалистам по системному анализу создавать алгоритмы, а программистам — разбираться в принципах работы программы. UML является мощным инструментом, позволяющим сделать процесс программирования более легким и эффективным. Мы проводим краткое ознакомление с UML в главе 1, а более специфические вопросы, связанные с UML, рассматриваем в других главах книги. Каждое новое средство UML вводится на рассмотрение в том месте, где оно становится полезным для иллюстрации разделов ООП. Таким образом, у вас появляется возможность, не прилагая лишних усилий, освоить концепции языка UML, который одновременно будет способствовать более эффективному усвоению C++.

Языки и платформы разработки

Из всех объектно-ориентированных языков C++ является наиболее широко употребительным. Язык Java, представляющий собой последнюю разработку в области объектно-ориентированных языков, лишен таких составляющих, как указатели, шаблоны и множественное наследование, что сделало его менее мощным и гибким по сравнению с C++ (синтаксис языка Java очень похож на синтаксис C++, поэтому знания относительно C++ с успехом могут быть применены при программировании на Java).

Некоторые другие объектно-ориентированные языки, например, C#, также успешно развиваются, однако их распространение в значительной степени уступает C++.

До последнего времени язык C++ развивался вне рамок стандартизации. Это означало, что каждый производитель компиляторов по-своему реализовывал отдельные нюансы языка. Тем не менее комитет по стандартам языка C++ организации ANSI/ISO разработал документ, ныне известный под названием *Стандартного C++*. (ANSI является сокращением от английского названия Американского Национального Института Стандартов, а ISO — от Международной Организации Стандартов.) Стандартный C++ включает в себя много дополнительных возможностей, например *стандартную библиотеку шаблонов (STL)*.

В этой книге мы будем придерживаться стандартного C++ за редкими исключениями, которые будут специальным образом оговорены.

Наиболее популярной средой разработки для C++ является продукт, совместно разработанный компаниями Microsoft и Borland и предназначенный для работы под управлением операционных систем Microsoft Windows. Мы постарались создать примеры программ для этой книги таким образом, чтобы они поддерживались как компиляторами Microsoft, так и компиляторами Borland (более детальную информацию о компиляторах можно найти в приложении В «Microsoft Visual C++» и приложении Г «Borland C++ Builder»).

Для чего нужна эта книга

Эта книга предназначена для изучения объектно-ориентированного программирования на языке C++ с помощью компиляторов фирм Microsoft и Borland. В ней также рассматриваются методы разработки программного обеспечения и графический язык UML. Книга будет полезна как профессиональным программистам, так и студентам, изучающим программирование, а также всем тем, кто испытывает интерес к данной тематике.

Новые концепции

Объектно-ориентированное программирование содержит несколько концепций, которые могут быть незнакомы тем, кто практикует программирование на таких традиционных языках, как Pascal, Basic и C. К этим концепциям относятся *классы*, *наследование* и *полиморфизм*, составляющие основу объектно-ориентированного подхода. Тем не менее, рассматривая специфичные вопросы объектно-ориентированных языков, довольно легко уйти в сторону от этих понятий. Многие книги обрушивают на читателя море деталей относительно возможностей языка, не уделяя внимания тому, почему эти возможности существуют. В этой книге нюансы языка рассматриваются в связи с базовыми концепциями.

Последовательность изложения материала

В этой книге мы начинаем с рассмотрения самых простых примеров, постепенно усложняя их и заканчивая полноценными объектно-ориентированными приложениями. Интенсивность подачи нового материала такова, что у читателя есть возможность без спешки усваивать пройденный материал. Для облегчения усвоения нового материала мы постарались снабдить его максимальным количеством наглядных иллюстраций. В конце каждой главы приведены контрольные вопросы и упражнения, что позволяет использовать книгу в качестве практического пособия для студентов. Ответы на вопросы и упражнения, помеченные знаком *, приведены в приложении Ж. Упражнения имеют различную степень сложности.

Знания, необходимые для чтения этой книги

Материал, изложенный в этой книге, доступен даже тем, кто изучает программирование «с нуля». Однако наличие опыта программирования на таких языках, как Visual Basic, не будет помехой при чтении.

Многие учебники по C++ рассчитаны на то, что читатель уже знаком с языком C. Данная книга в этом смысле является исключением. Изучение C++ в этой книге начинается «с чистого листа», и отсутствие знакомства с языком C никак не скажется на эффективности изучения C++. Читатели, знакомые с C, наверняка обратят внимание на то, как мало существует различий между C и C++.

Желательно, чтобы читатель этой книги имел представление об основных операциях Microsoft Windows, таких, как запуск приложений и копирование файлов.

Техническое и программное обеспечение

Для работы с программами вам потребуется компилятор языка C++. Для примеров из этой книги подойдут компиляторы Microsoft C++ и Borland C++.

В приложении В этой книги можно найти подробную информацию о работе с компилятором фирмы Microsoft, а в приложении Г — с компилятором фирмы Borland. Другие компиляторы, рассчитанные на стандартный C++, будут безошибочно воспринимать большинство примеров в том виде, в каком они приведены в книге.

Для того чтобы выбранный компилятор мог работать на вашем компьютере, последний должен обладать достаточным свободным дисковым пространством, объемом оперативной памяти и скоростью работы процессора. Эти параметры можно узнать из руководств по эксплуатации соответствующих устройств.

Консольные программы

В этой книге содержится большое количество примеров программ. Все программы являются консольными, то есть выполняющимися в текстовом окне компилятора, либо непосредственно в управляющем окне MS DOS. Это сделано для того, чтобы избежать сложностей, возникающих при работе с полноценными графическими приложениями Windows.

Исходные тексты программ

Исходные тексты программ доступны на сервере http://www.samspublishing.com.

Упражнения

Каждая глава содержит около 12 упражнений, предполагающих создание законченной программы на С++. Решения первых 3–4 упражнений приведены в приложении Ж. Остальные упражнения рассчитаны на самостоятельное выполнение (читателям, занимающимся преподавательской деятельностью, рекомендуется прочитать раздел «Преподавателям» данного вступления).

Проще, чем кажется

Говорят, что С++ трудно выучить. На самом деле С++ имеет немало общего с другими языками программирования, за исключением нескольких новых идей. Эти идеи интересны и увлекательны, поэтому их изучение едва ли покажется скучным. Кроме того, об этих идеях необходимо иметь представление хотя бы потому, что они составляют часть «культуры программирования». Мы надеемся, что эта книга поможет вам разобраться как в общих концепциях, так и в специфике программирования на С++.

Преподавателям

Возможно, преподавателям и другим читателям этой книги, владеющим С и С++, будет интересно более подробно ознакомиться с концепцией того подхода, который мы предлагаем при изучении С++.

Стандартный С++

Весь программный код в этой книге удовлетворяет требованиям стандартного С++ за незначительными исключениями, которые обусловлены некорректной работой компиляторов. Отдельная глава посвящена STL (стандартной библиотеке шаблонов), являющейся частью стандартного С++.

Унифицированный язык моделирования (UML)

В предыдущем издании книги мы рассказывали о UML в последней главе. В настоящем издании изучение UML проходит в различных главах книги, и, как правило, там, где это необходимо. Например, диаграммы классов UML изучаются одновременно с взаимодействием различных классов, а обобщение — в связи с понятием наследования.

Глава 1 «Общие сведения» включает список тем, касающихся UML, с указанием их расположения в книге.

Средства разработки программного обеспечения

Разработка прикладного программного обеспечения становится все более и более важным аспектом программирования. Увы, но зачастую процесс создания объектно-ориентированной программы остается для студентов загадкой. Это послужило для нас поводом включить в книгу специальную главу, касающуюся разработки программного обеспечения, где основное внимание уделено объектно-ориентированному программированию.

Различия между C и C++

В настоящее время существует лишь небольшое число высших и средних специальных учебных заведений, в которых перед изучением C++ студентов сначала обучают языку C. Мы также не считаем такой подход правильным, поскольку, по нашему мнению, языки C и C++ полностью независимы. В синтаксическом плане C и C++ очень схожи; более того, C является подмножеством C++. Однако установившееся мнение о схожести самих языков является исторически ошибочным, поскольку подходы к написанию программ на C и C++ кардинальным образом различаются.

C++ завоевал большую популярность, чем C, потому что что стал мощным инструментом для разработки сложного программного обеспечения. Это послужило причиной для того, чтобы не рассматривать изучение C как обязательное перед изучением C++. Те студенты, которые знакомы с языком C, могут опустить часть материала, изложенного в книге, однако значительная часть материала окажется для них новой.

Оптимальный порядок изучения ООП

Мы могли бы начать эту книгу с изучения процедурно-ориентированной концепции программирования, присущей языкам C и C++, и лишь после этого приступить к ООП, однако для нас является более важным приступить к объектно-ориентированному программированию как можно быстрее. Таким образом, нашей задачей является рассмотрение основы процедурного программирования, а затем — непосредственно изучение классов. Уже начальные главы этой книги демонстрируют применение языка C++, в корне отличающееся от C.

Некоторые вопросы рассматриваются в этой книге несколько раньше, чем это принято в другой литературе по языку C: например, рассмотрение структур предшествует рассмотрению классов. Причиной для этого является то, что класс представляет собой не что иное, как синтаксическое расширение структуры, и, следовательно, структура является ключом к пониманию C++.

Такие элементы языка, как указатели, рассматриваются несколько позже, чем в большинстве книг по языку C. Указатели не являются необходимыми при изучении основ ООП и в то же время, как правило, представляют трудность для понимания. Таким образом, обсуждение указателей откладывается до тех

пор, пока не будут рассмотрены основные концепции объектно-ориентированного программирования.

Нововведения в C++

Некоторые средства языка C были заменены на аналоги из языка C++. Например, основные функции ввода/вывода языка C printf() и scanf() в C++ практически не используются, поскольку cin и cout более удобны. Поэтому мы опускаем описания этих и некоторых других функций. По аналогичной причине вместо макроопределения #define, типичного для C, практически везде используется спецификатор const и встроенные функции C++.

Избыточные возможности

Поскольку основное внимание в этой книге сконцентрировано на объектно-ориентированном программировании, мы можем исключить из рассмотрения те средства языка C, которые редко используются и не имеют отношения к объектно-ориентированному программированию. Примером одного из таких средств являются побитовые операции языка C. Мы не будем детально рассматривать эти средства, а лишь кратко упомянем их.

Итак, результатом нашей работы является книга, неспешно, но динамично вводящая читателя в курс объектно-ориентированного программирования и его практического применения.

Упражнения

В книге не приводятся решения для тех упражнений, которые не помечены знаком *, но их можно найти на сайте издательства, как и примеры, встречающиеся в этой книге.

Упражнения каждой главы различаются между собой по степени сложности. Первые по порядку упражнения обычно являются легкими, а в конце списка располагаются наиболее сложные упражнения. Это будет удобно для преподавателей, которые получают возможность обеспечивать своих студентов заданиями, соответствующими уровню их подготовки.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства http://www.piter.com.

С текстами программ из этой книги вы можете ознакомиться по адресу http://www.piter.com/download.

Глава 1
Общие сведения

- Зачем нужно объектно-ориентированное программирование
- Характеристики объектно-ориентированных языков
- С++ и С
- Изучение основ
- Универсальный язык моделирования (UML)

Изучив эту книгу, вы получите основные навыки создания программ на языке С++, поддерживающем *объектно-ориентированное программирование* (ООП). Для чего нужно ООП? Каковы преимущества ООП перед такими традиционными языками программирования, как Pascal, С или BASIC? Что является основой ООП? В ООП существует две ключевые концепции — *объекты* и *классы*. Каков смысл этих двух терминов? Как связаны между собой языки С и С++?

Эта глава посвящена рассмотрению перечисленных вопросов, а также обзору средств языка, о которых пойдет речь в других главах книги. Не беспокойтесь, если материал, изложенный в этой главе, покажется вам чересчур абстрактным. Все методы и концепции, которые мы упомянем, будут подробно рассмотрены в последующих главах книги.

Для чего нужно объектно-ориентированное программирование?

Развитие объектно-ориентированного метода обусловлено ограниченностью других методов программирования, разработанных ранее. Чтобы лучше понять и оценить значение ООП, необходимо разобраться, в чем состоит эта ограниченность и каким образом она проявляется в традиционных языках программирования.

Процедурные языки

С, Pascal, FORTRAN и другие сходные с ними языки программирования относятся к категории *процедурных языков*. Каждый оператор такого языка является

указанием компьютеру совершить некоторое действие, например принять данные от пользователя, произвести с ними определенные действия и вывести результат этих действий на экран. Программы, написанные на процедурных языках, представляют собой последовательности инструкций.

Для небольших программ не требуется дополнительной внутренней организации (часто называемой термином *парадигма*). Программист создает перечень инструкций, а компьютер выполняет действия, соответствующие этим инструкциям.

Деление на функции

Когда размер программы велик, список команд становится слишком громоздким. Очень небольшое число программистов способно удерживать в голове более 500 строк программного кода, если этот код не разделен на более мелкие логические части. *Функция* является средством, облегчающим восприятие при чтении текста программы (термин *функция* употребляется в языках С и С++; в других языках программирования это же понятие называют подпрограммой или процедурой). Программа, построенная на основе процедурного метода, разделена на функции, каждая из которых в идеальном случае выполняет некоторую законченную последовательность действий и имеет явно выраженные связи с другими функциями программы.

Можно развить идею разбиения программы на функции, объединив несколько функций в *модуль* (зачастую модуль представляет собой отдельный файл). При этом сохраняется процедурный принцип: программа делится на несколько компонентов, каждый из которых представляет собой набор инструкций.

Деление программы на функции и модули является основой структурного программирования. Структурное программирование представляет собой нечто не вполне определенное, однако в течение нескольких десятков лет, пока не была разработана концепция объектно-ориентированного программирования, оно оставалось важным способом организации программ.

Недостатки структурного программирования

В непрекращающемся процессе роста и усложнения программ стали постепенно выявляться недостатки структурного подхода к программированию. Возможно, вам приходилось слышать «страшные истории» о том, как происходит работа над программным проектом, или даже самим участвовать в создании такого проекта: задача оказывается сложнее, чем казалось, сроки сдачи проекта переносятся. Все новые и новые программисты привлекаются для работы, что резко увеличивает расходы. Окончание работы вновь переносится, и в результате проект терпит крах.

Проанализировав причины столь печальной судьбы многих проектов, можно прийти к выводу о недостаточной мощи структурного программирования: как бы эффективно ни применялся структурный подход, он не позволяет в достаточной степени упростить большие сложные программы.

В чем же недостаток процедурно-ориентированных языков? Существует две основные проблемы. Первая заключается в неограниченности доступа функций к глобальным данным. Вторая состоит в том, что разделение данных и функций, являющееся основой структурного подхода, плохо отображает картину реального мира.

Давайте рассмотрим эти недостатки на примере программы складского учета. В такой программе глобальными данными являются записи в учетной книге. Различные функции будут получать доступ к этим данным для выполнения операций создания новой записи, вывода записи на экран, изменения существующей записи и т. д.

Неконтролируемый доступ к данным

В процедурной программе, написанной, к примеру, на языке C, существует два типа данных. *Локальные данные* находятся внутри функции и предназначены для использования исключительно этой функцией. Например, в программе складского учета функция, осуществляющая вывод записи на экран, может использовать локальные данные для хранения информации о выводимой записи. Локальные данные функции недоступны никому, кроме самой функции, и не могут быть изменены другими функциями.

Если существует необходимость совместного использования одних и тех же данных несколькими функциями, то данные должны быть объявлены как *глобальные*. Это, как правило, касается тех данных программы, которые являются наиболее важными. Примером здесь может служить уже упомянутая учетная книга. Любая функция имеет доступ к глобальным данным (мы не рассматриваем случай группирования функций в модули). Схема, иллюстрирующая концепцию локальных и глобальных данных, приведена на рис. 1.1.

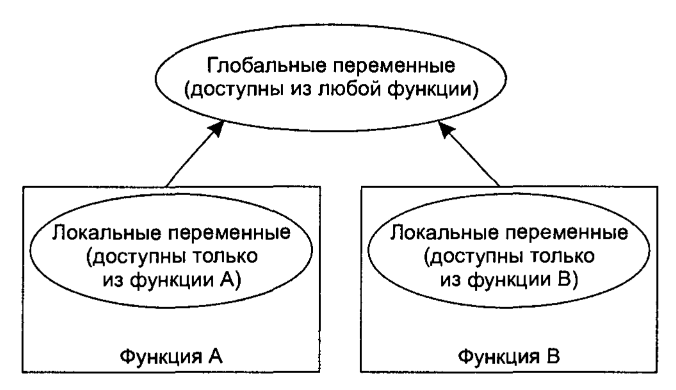


Рис. 1.1. Глобальные и локальные переменные

Большие программы обычно содержат множество функций и глобальных переменных. Проблема процедурного подхода заключается в том, что число возможных связей между глобальными переменными и функциями может быть очень велико, как показано на рис. 1.2.

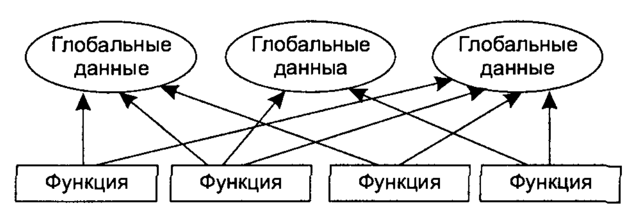


Рис. 1.2. Процедурный подход

Большое число связей между функциями и данными, в свою очередь, также порождает несколько проблем. Во-первых, усложняется структура программы. Во-вторых, в программу становится трудно вносить изменения. Изменение структуры глобальных данных может потребовать переписывания всех функций, работающих с этими данными. Например, если разработчик программы складского учета решит сделать код продукта не 5-значным, а 12-значным, то будет необходимо изменить соответствующий тип данных с short на long. Это означает, что во все функции, оперирующие кодом продукта, должны быть внесены изменения, позволяющие обрабатывать данные типа long. Можно привести аналогичный бытовой пример, когда в супермаркете изменяется расположение отделов, и покупателям приходится соответствующим образом менять свой привычный путь от одного отдела к другому.

Когда изменения вносятся в глобальные данные больших программ, бывает непросто быстро определить, какие функции необходимо скорректировать. Даже в том случае, когда это удается сделать, из-за многочисленных связей между функциями и данными исправленные функции начинают некорректно работать с другими глобальными данными. Таким образом, любое изменение влечет за собой далеко идущие последствия.

Моделирование реального мира

Вторая, более важная, проблема процедурного подхода заключается в том, что отделение данных от функций оказывается малопригодным для отображения картины реального мира. В реальном мире нам приходится иметь дело с физическими объектами, такими, например, как люди или машины. Эти объекты нельзя отнести ни к данным, ни к функциям, поскольку реальные вещи представляют собой совокупность *свойств* и *поведения*.

Свойства

Примерами свойств (иногда называемых характеристиками) для людей могут являться цвет глаз или место работы; для машин — мощность двигателя и количество дверей. Таким образом, свойства объектов равносильны данным в программах: они имеют определенное значение, например *голубой* для цвета глаз или *4* для количества дверей автомобиля.

Поведение

Поведение — это некоторая реакция объекта в ответ на внешнее воздействие. Например, ваш босс в ответ на просьбу о повышении может дать ответ «да» или

«нет». Если вы нажмете на тормоз автомобиля, это повлечет за собой его остановку. Ответ и остановка являются примерами поведения. Поведение сходно с функцией: вы вызываете функцию, чтобы совершить какое-либо действие (например, вывести на экран учетную запись), и функция совершает это действие.

Таким образом, ни отдельно взятые данные, ни отдельно взятые функции не способны адекватно отобразить объекты реального мира.

Объектно-ориентированный подход

Основополагающей идеей объектно-ориентированного подхода является объединение *данных* и *действий, производимых над этими данными*, в единое целое, которое называется *объектом.*

Функции объекта, называемые в C++ *методами* или *функциями-членами*, обычно предназначены для доступа к данным объекта. Если необходимо считать какие-либо данные объекта, нужно вызвать соответствующий метод, который выполнит считывание и возвратит требуемое значение. Прямой доступ к данным невозможен. Данные сокрыты от внешнего воздействия, что защищает их от случайного изменения. Говорят, что данные и методы *инкапсулированы.* Термины *сокрытие* и *инкапсуляция* данных являются ключевыми в описании объектно-ориентированных языков.

Если необходимо изменить данные объекта, то, очевидно, это действие также будет возложено на методы объекта. Никакие другие функции не могут изменять данные класса. Такой подход облегчает написание, отладку и использование программы.

Типичная программа на языке C++ состоит из совокупности объектов, взаимодействующих между собой посредством вызова методов друг друга. Структура программы на C++ приводится на рис. 1.3.

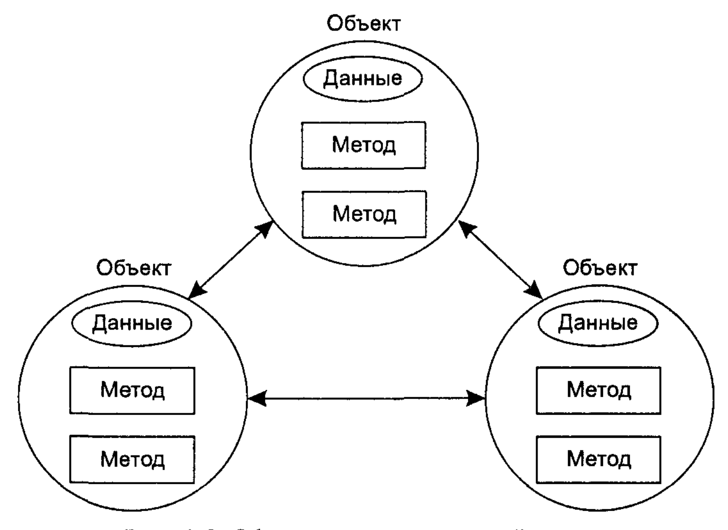


Рис. 1.3. Объектно-ориентированный подход

Аналогия

Возможно, вы представляете себе объекты чем-то вроде отделов компании — бухгалтерии, продаж, кадров и т. п. Деление на отделы является важной частью структурной организации фирмы. В большинстве компаний (за исключением небольших) в обязанности отдельного сотрудника не входит решение одновременно кадровых, торговых и учетно-бухгалтерских вопросов. Обязанности четко распределяются между подразделениями, и у каждого подразделения имеются данные, с которыми оно работает: у бухгалтерии — заработная плата, у отдела продаж — сведения, касающиеся торговли, у отдела кадров — персональная информация о сотрудниках и т. д.

Сотрудники каждого отдела производят операции только с теми данными, которые относятся к данному отделу. Подобное разделение обязанностей позволяет легче следить за деятельностью компании и контролировать ее, а также поддерживать целостность информационного пространства компании. Например, бухгалтерия несет ответственность за информацию, касающуюся зарплат. Если у менеджера по продажам возникнет необходимость узнать суммарный оклад сотрудников компании за июль, то ему не нужно будет идти в бухгалтерию и рыться в картотеках; ему достаточно послать запрос компетентному лицу, которое должно найти нужную информацию, обработать ее и выслать ответ на запрос. Такая схема позволяет обеспечить правильную обработку данных, а также ее защиту от возможного воздействия посторонних лиц. Подобная структура компании изображена на рис. 1.4. Подобным же образом объекты создают такую организацию программы, которая обеспечивает целостность ее данных.

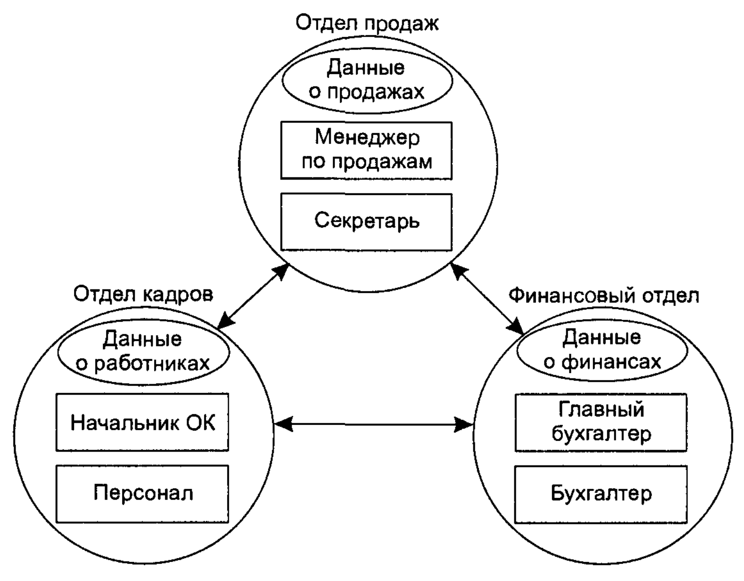


Рис. 1.4. Корпоративный подход

ООП: подход к организации программы

Объектно-ориентированное программирование никак не связано с процессом выполнения программы, а является лишь способом ее организации. Большая часть операторов C++ идентична операторам процедурных языков, в частности языка С. Внешне метод класса в C++ очень похож на обычную функцию языка С, и только по контексту программы можно определить, является ли функция частью процедурной С-программы или объектно-ориентированной программы на C++.

Характеристики объектно-ориентированных языков

Здесь мы рассмотрим несколько основных элементов, входящих в состав объектно-ориентированных языков, в частности в состав языка C++.

Объекты

Когда вы подходите к решению задачи с помощью объектно-ориентированного метода, то вместо проблемы разбиения задачи на функции вы сталкиваетесь с проблемой разбиения ее на объекты. Мышление в терминах объектов оказывается гораздо более простым и наглядным, чем в терминах функций, поскольку программные объекты схожи с объектами реального мира. Более подробно данный вопрос рассмотрен в главе 16 «Разработка объектно-ориентированного программного обеспечения».

Что должно представляться в программе в виде объектов? Окончательный ответ на этот вопрос может дать только ваше воображение, однако приведем несколько советов, которые могут оказаться полезными:

- Физические объекты.
  - Автомобили при моделировании уличного движения.
  - Схемные элементы при моделировании цепи электрического тока.
  - Страны при создании экономической модели.
  - Самолеты при моделировании диспетчерской системы.
- Элементы интерфейса.
  - Окна.
  - Меню.
  - Графические объекты (линии, прямоугольники, круги).
  - Мышь, клавиатура, дисковые устройства, принтеры.
- Структуры данных.
  - Массивы.
  - Стеки.

    - Связанные списки.
    - Бинарные деревья.
- Группы людей.
    - Сотрудники.
    - Студенты.
    - Покупатели.
    - Продавцы.
- Хранилища данных.
    - Описи инвентаря.
    - Списки сотрудников.
    - Словари.
    - Географические координаты городов мира.
- Пользовательские типы данных.
    - Время.
    - Величины углов.
    - Комплексные числа.
    - Точки на плоскости.
- Участники компьютерных игр.
    - Автомобили в гонках.
    - Позиции в настольных играх (шашки, шахматы).
    - Животные в играх, связанных с живой природой.
    - Друзья и враги в приключенческих играх.

Соответствие между программными и реальными объектами является следствием объединения данных и функций. Получающиеся в результате такого объединения объекты в свое время произвели фурор, ведь ни одна программная модель, созданная на основе процедурного подхода, не отражала существующие вещи столь точно, как это удалось сделать с помощью объектов.

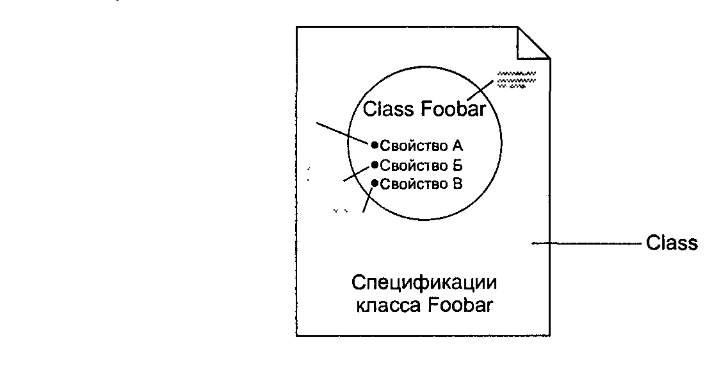
Классы

Когда мы говорим об объектах, мы говорим, что они являются экземплярами *классов*. Что это означает? Рассмотрим следующую аналогию. Практически все компьютерные языки имеют стандартные типы данных; например, в C++ есть целый тип int. Мы можем определять переменные таких типов в наших программах:

```
int day:
int count;
int divisor:
int answer;
```

Подобным же образом мы можем определять объекты класса, как показано на рис. 1.5. Класс является своего рода формой, определяющей, какие данные

и функции будут включены в объект класса. При объявлении класса не создаются никакие объекты этого класса, по аналогии с тем, что существование типа int еще не означает существование переменных этого типа.



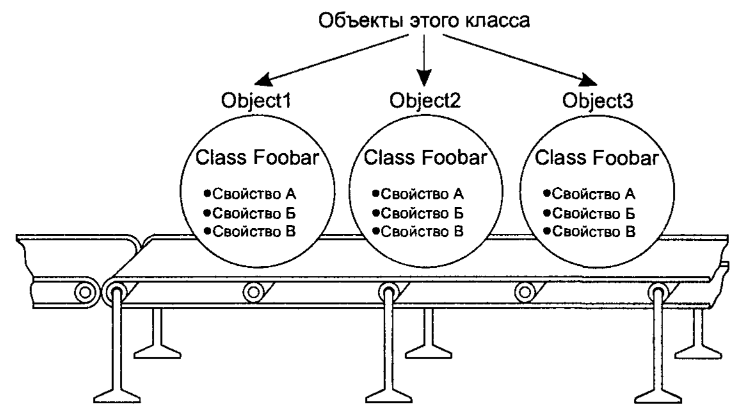


Рис. 1.5. Класс и его объекты

Таким образом, класс является описанием совокупности сходных между собой объектов. Это соответствует нестрогому в техническом смысле пониманию термина «класс»: например, Prince, Sting и Madonna относятся к классу рок-музыкантов. Не существует конкретного человека с именем «рок-музыкант», однако люди со своими уникальными именами являются объектами этого класса, если они обладают определенным набором характеристик. Объект класса часто также называют *экземпляром* класса.

Наследование

Понятие класса приводит нас к понятию *наследования*. В повседневной жизни мы часто сталкиваемся с разбиением классов на подклассы: например, класс *животные* можно разбить на подклассы *млекопитающие*, *земноводные*, *насекомые*, *птицы* и т. д. Класс *наземный транспорт* делится на классы *автомобили*, *грузовики*, *автобусы*, *мотоциклы* и т. д.

Принцип, положенный в основу такого деления, заключается в том, что каждый подкласс обладает свойствами, присущими тому классу, из которого выделен данный подкласс. Автомобили, грузовики, автобусы и мотоциклы обладают колесами и мотором, являющимися характеристиками наземного транспорта. Кроме тех свойств, которые являются общими у данных класса и подкласса, подкласс может обладать и собственными свойствами: например, автобусы имеют большое число посадочных мест для пассажиров, в то время как грузовики обладают значительным пространством и мощностью для перевозки тяжеловесных грузов и т. д. Иллюстрация этой идеи приведена на рис. 1.6.

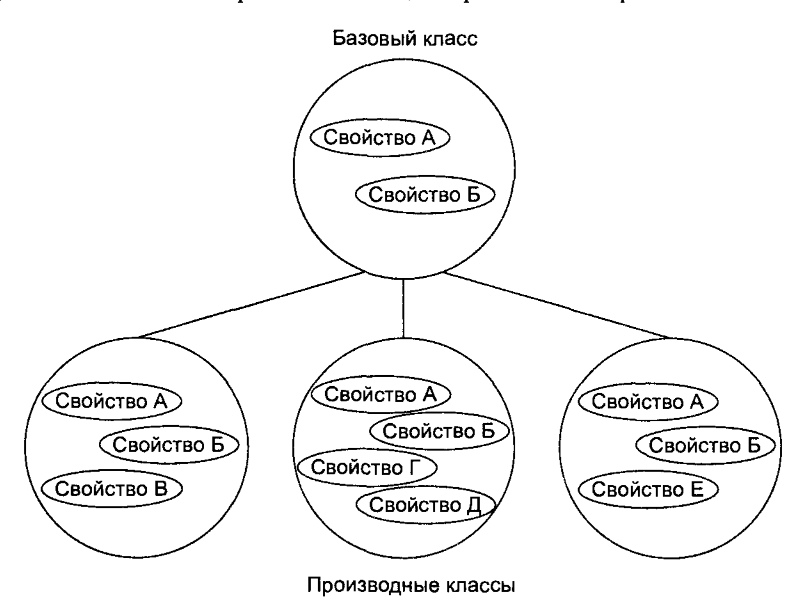


Рис. 1.6. Наследование

Подобно этому, в программировании класс также может породить множество подклассов. В C++ класс, который порождает все остальные классы, называется *базовым классом*; остальные классы наследуют его свойства, одновременно обладая собственными свойствами. Такие классы называются *производными*.

Не проводите ложных аналогий между отношениями «объект—класс» и «базовый класс — производный класс»! Объекты, существующие в памяти компьютера, являются воплощением свойств, присущих классу, к которому они принадлежат. Производные классы имеют свойства как унаследованные от базового класса, так и свои собственные.

Наследование можно считать аналогом использования функций в процедурном подходе. Если мы обнаружим несколько функций, совершающих похожие действия, то мы извлечем из них идентичные части и вынесем их в отдельную функцию. Тогда исходные функции будут одинаковым образом вызывать свою общую часть, и в то же время в каждой из них будут содержаться свои собственные инструкции. Базовый класс содержит элементы, общие для группы про-

изводных классов. Роль наследования в ООП такая же, как и у функций в процедурном программировании, — сократить размер кода и упростить связи между элементами программы.

Повторное использование кода

Разработанный класс может быть использован в других программах. Это свойство называется возможностью *повторного использования кода*. Аналогичным свойством в процедурном программировании обладают библиотеки функций, которые можно включать в различные программные проекты.

В ООП концепция наследования открывает новые возможности повторного использования кода. Программист может взять существующий класс, и, ничего не изменяя, добавить в него свои элементы. Все производные классы унаследуют эти изменения, и в то же время каждый из производных классов можно модифицировать отдельно.

Предположим, что вы (или кто-то другой) разработали класс, представляющий систему меню, аналогичную графическому интерфейсу Microsoft Windows или другому графическому интерфейсу пользователя (GUI). Вы не хотите изменять этот класс, но вам необходимо добавить возможность установки и снятия флажков. В этом случае вы создаете новый класс, наследующий все свойства исходного класса, и добавляете в него необходимый код.

Легкость повторного использования кода уже написанных программ является важным достоинством ООП. Многие компании утверждают, что возможность включать в новые версии программного обеспечения коды программ более старых версий благоприятно сказывается на прибыли, приносимой последними. Более подробно этот вопрос будет обсуждаться в других главах книги.

Пользовательские типы данных

Одним из достоинств объектов является то, что они дают пользователю возможность создавать свои собственные типы данных. Представьте себе, что вам необходимо работать с объектами, имеющими две координаты, например x и y. Вам хотелось бы совершать обычные арифметические операции над такими объектами, например:

```
position1 = position2 + origin
```

где переменные position1, position2 и origin представляют собой наборы из двух координат. Описав класс, включающий в себя пару координат, и объявив объекты этого класса с именами position1, position2 и origin, мы фактически создадим новый тип данных. В C++ имеются средства, облегчающие создание подобных пользовательских типов данных.

Полиморфизм и перегрузка

Обратите внимание на то, что операции присваивания = и сложения + для типа position должны выполнять действия, отличающиеся от тех, которые они выпол-

няют для объектов стандартных типов, например int. Объекты position1 и прочие не являются стандартными, поскольку определены пользователем как принадлежащие классу position. Как же операторы = и + распознают, какие действия необходимо совершить над операндами? Ответ на этот вопрос заключается в том, что мы сами можем задать эти действия, сделав нужные операторы методами класса position.

Использование операций и функций различным образом в зависимости от того, с какими типами величин они работают, называется *полиморфизмом*. Когда существующая операция, например = или +, наделяется возможностью совершать действия над операндами нового типа, говорят, что такая операция является *перегруженной*. Перегрузка представляет собой частный случай полиморфизма и является важным инструментом ООП.

C++ и C

C++ унаследовал возможности языка C. Строго говоря, C++ является расширением языка C: любая конструкция на языке C является корректной в C++; в то же время обратное неверно. Наиболее значительные нововведения, присутствующие в C++, касаются классов, объектов и ООП (первоначальное название языка C++ — «C с классами»). Тем не менее имеются и другие усовершенствования, связанные со способами организации ввода/вывода и написанием комментариев. Иллюстрация отношения между языками C и C++ приведена на рис. 1.7.

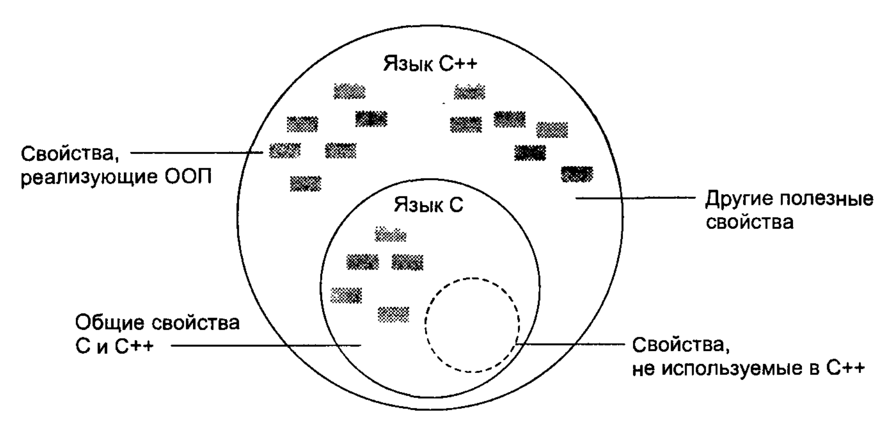


Рис. 1.7. Отношения между C и C++

На практике существует гораздо больше различий между C и C++, чем может показаться вначале. Несмотря на то что можно писать программы на C++, внешне напоминающие программы на C, вряд ли кому-то придет в голову так поступать. Программисты, использующие C++, не только пользуются преимуществами этого языка перед C, но и по-новому используют возможности, унаследованные от C. Если вы знакомы с C, значит, у вас уже имеются некоторые

знания относительно C++, но все же вероятно, что значительная часть материала окажется для вас новой.

Изучение основ

Наша задача состоит в том, чтобы как можно быстрее научить вас создавать объектно-ориентированные программы. Поскольку, как мы уже говорили, значительная часть языка C++ унаследована от C, то даже при объектно-ориентированной структуре программы ее основу составляют «старомодные» процедурные средства. Поэтому в главах 2–5 рассматриваются традиционные средства программирования языка C++, в основном унаследованные от C: действия с переменными и ввод/вывод, организация циклов и переходов, работа с функциями. Там же будут рассмотрены структуры, синтаксис которых совпадает с синтаксисом классов.

Если вы уже знакомы с языком C, то у вас, возможно, возникнет желание пропустить эти главы. Тем не менее, не стоит забывать о различиях между C и C++, которые могут как лежать на поверхности, так и быть незаметными при невнимательном чтении. Поэтому мы советуем читателю бегло просматривать тот материал, который ему известен, а основное внимание сконцентрировать на различиях C и C++.

Детальное рассмотрение ООП начнется в главе 6 «Объекты и классы». Все примеры, начиная с этой главы, будут иметь объектно-ориентированную структуру.

Универсальный язык моделирования (UML)

UML можно условно называть графическим языком, предназначенным для моделирования компьютерных программ. Под моделированием понимается создание наглядной визуальной интерпретации чего-либо. UML позволяет создавать подобную интерпретацию программ высокоуровневой организации.

Родоначальниками UML стали три независимых языка моделирования, создателями которых были соответственно Гради Буч, Джеймс Рэмбо и Ивар Джекобсон. В конце 90-х годов они объединили свои разработки, в результате чего получили продукт под названием *универсальный язык моделирования (UML)*, который был одобрен OMG — консорциумом компаний, определяющих промышленные стандарты.

Почему UML необходим? Во-первых, потому, что бывает трудно установить взаимоотношение частей большой программы между собой посредством анализа ее кода. Как мы уже видели, объектно-ориентированное программирование является более прогрессивным, чем процедурное. Но даже при этом подходе для того, чтобы разобраться в действиях программы, необходимо как минимум представлять себе содержание ее кода.

Проблема изучения кода состоит в том, что он очень детализован. Гораздо проще было бы взглянуть на его общую структуру, отображающую только основные части программы и их взаимодействие. UML обеспечивает такую возможность.

Наиболее важным средством UML является набор различных видов диаграмм. Диаграммы классов иллюстрируют отношения между различными классами, диаграммы объектов — между отдельными объектами, диаграммы связей отражают связь объектов во времени и т. д. Все эти диаграммы, по сути, отражают взгляды на программу и ее действия с различных точек зрения.

Кроме иллюстрирования структуры программы, UML имеет немало других полезных возможностей. В главе 16 пойдет речь о том, как с помощью UML разработать первоначальную структуру программы. Фактически UML можно использовать на всех этапах создания проекта — от разработки до документирования, тестирования и поддержки.

Тем не менее не стоит рассматривать UML как средство разработки программного обеспечения. UML является лишь средством для иллюстрирования разрабатываемого проекта. Несмотря на возможность применения к любому типу языков, UML наиболее полезен в объектно-ориентированном программировании.

Как мы уже упомянули во введении, мы будем постепенно рассматривать новые средства UML по ходу изложения основного материала книги.

- Глава 1: введение в UML.
- Глава 8: диаграммы классов, ассоциации, возможности перемещения.
- Глава 9: обобщение, агрегация, композиция классов.
- Глава 10: диаграмма состояний и множественности.
- Глава 11: диаграммы объектов.
- Глава 13: более сложные диаграммы состояний.
- Глава 14: шаблоны, зависимости и стереотипы.
- Глава 16: варианты использования, диаграммы вариантов использования, диаграммы действий и диаграммы последовательностей.

Резюме

ООП является способом организации программы. Основное внимание при его изучении уделяется организации программы, а не вопросам написания кода. Главным компонентом объектно-ориентированной программы является объект, содержащий данные и функции для их обработки. Класс является формой или образцом для множества сходных между собой объектов.

Механизм наследования позволяет создавать новые классы на основе существующих классов, не внося изменений в последние. Порожденный класс наследует все данные и методы своего родителя, но имеет также и свои собственные. Наследование делает возможным повторное использование кода, то есть включение однажды созданного класса в любые другие программы.

C++ является расширением языка C, позволяющим реализовать концепцию ООП, а также включающим в себя некоторые дополнительные возможности. Часть средств языка C, несмотря на их поддержку C++, признаны устаревшими в контексте новых подходов к программированию и потому употребляются редко, как правило, заменяясь более новыми средствами C++. В результате различия между C и C++ оказываются более значительными, чем кажется на первый взгляд.

Универсальный язык моделирования (UML) является стандартизованным средством визуализации структуры и функционирования программы посредством диаграмм.

Идеи, обсуждавшиеся в этой главе, будут конкретизироваться по мере изучения C++. Возможно, в процессе чтения других глав книги вы ощутите необходимость вернуться к материалу, изложенному в этой главе.

Вопросы

Ответы на приведенные ниже вопросы можно найти в приложении Ж. Вопросы с альтернативными вариантами ответов могут иметь несколько верных ответов.

1. Языки Pascal, BASIC и C являются __________ языками, в то время как C++ является __________ языком.
2. В качестве образца по отношению к объекту выступает:
   а) метод;
   б) класс;
   в) операция;
   г) значение.
3. Двумя основными компонентами объекта являются __________ и функции, которые __________.
4. В C++ функцию, входящую в состав класса, называют:
   а) функция-член класса;
   б) оператор класса;
   в) функция класса;
   г) метод класса.
5. Защита данных от несанкционированного доступа другими функциями называется __________.
6. Какие из перечисленных ниже причин являются главными для использования объектно-ориентированных языков?
   а) возможность создания собственных типов данных;
   б) простота операторов объектно-ориентированных языков по сравнению с процедурными языками;

в) наличие средств для автокоррекции ошибок в объектно-ориентированных языках;

г) объектно-ориентированные программы легче концептуализируются.

7. _________ отображают объекты реального мира точнее, чем функции.

8. Истинно ли утверждение: программа на C++ в целом схожа с программой на C за исключением незначительных различий в кодировании.

9. Объединение данных и функций называется ____________.

10. Если язык обеспечивает возможность создания пользовательских типов данных, то говорят, что язык называется:

а) наследуемым;

б) инкапсулируемым;

в) перегруженным;

г) расширяемым.

11. Верно или неверно утверждение: двух операторов достаточно, чтобы легко отличить программу на C++ от программы на C.

12. Возможность выполнения оператором или функцией различных действий в зависимости от типа операндов называется ____________.

13. Операция, выполняющая заданные действия над пользовательским типом данных, называется:

а) полиморфической;

б) инкапсулированной;

в) классифицированной;

г) перегруженной.

14. Запоминание новых терминов языка C++:

а) является очень важным;

б) можно отложить «на потом»;

в) служит ключом к успеху и процветанию;

г) бессмысленно.

15. Универсальный язык моделирования — это:

а) программа для построения физических моделей;

б) средство визуализации организации программы;

в) результат объединения языков C++ и FORTRAN;

г) вспомогательное средство при разработке программного обеспечения.

Глава 2
Основы программирования на C++

- ✦ Что необходимо для работы
- ✦ Структура программы
- ✦ Вывод с использованием cout
- ✦ Директивы
- ✦ Комментарии
- ✦ Переменные целого типа
- ✦ Символьные переменные
- ✦ Ввод с помощью cin
- ✦ Вещественные типы
- ✦ Тип bool
- ✦ Манипулятор setw
- ✦ Таблица типов переменных
- ✦ Преобразования типов
- ✦ Арифметические операции
- ✦ Библиотечные функции

В каждом языке программирования существует основа, без знания которой невозможно написать даже самой простой программы. В этой главе мы займемся изучением такой основы для языка C++: структуры программы, переменных и базовых операций ввода/вывода. Здесь же мы рассмотрим и некоторые другие средства языка, такие, как комментарии, арифметические операции, операцию инкремента, преобразования типов и библиотечные функции.

Этот материал не представляет трудности для понимания, однако, как вы, возможно, заметите, стиль написания программ на языке C++ «жестче», чем на BASIC или Pascal. Пока вы в достаточной степени не привыкнете к такому сти-

лю, программы на C++ могут казаться вам по своей строгости сравнимыми с математическими формулами, однако это ощущение со временем пропадет. Без сомнения, уже небольшой опыт общения с C++ позволит вам чувствовать себя гораздо комфортнее, а другие языки программирования будут казаться чересчур прихотливыми и избыточными.

Что необходимо для работы

Как мы уже упоминали в введении, для работы с примерами из этой книги рекомендованы компиляторы фирм Microsoft и Borland. Подробная информация об этих компиляторах приведена в приложениях В и Г. Компиляторы преобразуют исходные коды программ в исполняемые файлы, которые можно запускать на вашем компьютере, как любые другие программы. Исходные файлы представляют собой текстовые файлы с расширением .CPP, которое используется и в листингах, приведенных в этой книге. Исполняемые файлы имеют расширение .EXE и могут запускаться как из компилятора, так и непосредственно в режиме MS DOS.

Программы, исполняемые с помощью компилятора Microsoft или из MS DOS, не требуют никакого дополнительного редактирования. Если же вы используете компилятор фирмы Borland, то требуется внести небольшие изменения в программу перед ее выполнением. О соответствующих действиях рассказано в приложении Г.

Структура программы

Рассмотрим первый, самый простой, пример программы на C++ под названием FIRST. Соответствующий файл исходного кода называется FIRST.CPP. Программа выводит сообщение на экран. Вот как она выглядит:

```
#include <iostream>
using namespace std:
int main()
{
  cout << "У каждой эпохи свой язык\n":
  return 0:
}
```

Несмотря на свой небольшой размер, этот пример демонстрирует типичную структуру программы на C++. Рассмотрим эту структуру в деталях.

Функции

Функции представляют собой основу, на которой строится любая программа C++. Программа FIRST состоит из единственной функции с названием main().

В состав этой функции не входят две первые строчки кода, начинающиеся со слов #include и using (мы раскроем смысл действий, заключенных в этих строках, чуть позже).

Как мы уже говорили в главе 1, функция может входить в состав класса, и в этом случае она называется *методом класса*. Тем не менее, функции могут существовать и отдельно от классов. Поскольку наших знаний сейчас недостаточно для того, чтобы говорить о классах, мы будем рассматривать лишь функции, которые являются независимыми, как функция main() в нашем примере.

Имена функций

Круглые скобки, идущие вслед за именем main(), являются отличительной чертой функций: если бы их не было, то компилятор не смог бы отличить имя переменной или другого элемента программы от имени функции. Когда мы будем использовать имена функций в объяснениях, мы всегда будем придерживаться соглашения, принятого в C++, и ставить круглые скобки после имени функции. Позже мы увидим, что в скобках могут указываться аргументы функций — имена переменных, значения которых программа передает в функцию.

Слово int, предваряющее имя функции, указывает на то, что эта функция возвращает значение типа int. Пока не стоит ломать голову над смыслом этой фразы: мы рассмотрим типы данных в этой главе, а типы значений, возвращаемых функциями, — в главе 5.

Тело функции

Тело функции заключено в фигурные скобки. Фигурные скобки играют ту же роль, что и ключевые слова BEGIN и END, встречающиеся в некоторых других языках программирования: они определяют границы блока операторов программы. Фигурные скобки, обрамляющие тело функции, обязательны. В нашем примере тело функции состоит всего лишь из двух операторов: один из них начинается словом cout, другой — словом return. Разумеется, функция может включать в себя и большее число операторов.

Функция main()

Когда программа на языке C++ запускается на выполнение, первым исполняемым оператором становится первый оператор функции main() (по крайней мере, это справедливо для консольных программ). Программа может состоять из множества функций, классов и прочих элементов, но при ее запуске управление всегда передается функции main(). Если в программе не содержится функции с именем main(), то при попытке запустить такую программу будет выведено сообщение об ошибке.

В большинстве программ, написанных на C++, реальные действия, выполняемые этими программами, сводятся к вызову функцией main() методов различных объектов. Кроме того, функция main() может вызывать другие, независимые функции. Иллюстрация вышесказанного приведена на рис. 2.1.

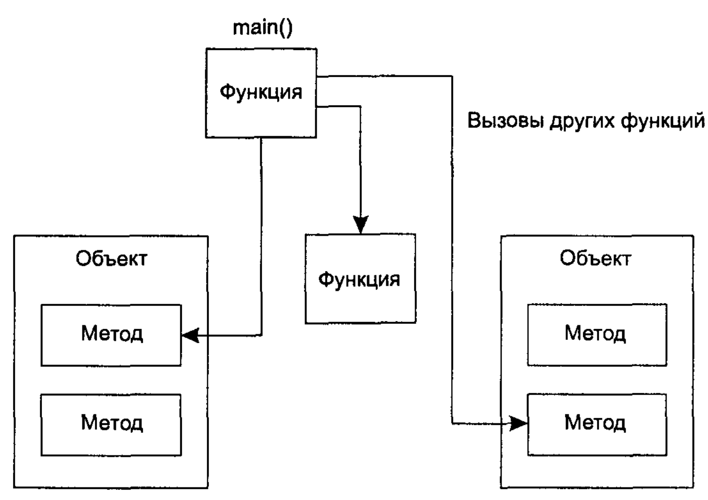


Рис. 2.1. Объекты, функции и main()

Операторы

Оператор является структурной единицей программы на C++. В программе FIRST содержатся 2 оператора:

```
cout << "У каждой эпохи свой язык\n";
return 0;
```

Первый оператор задает действие по выводу фразы, заключенной в кавычки, на экран. Вообще говоря, большинство операторов являются указаниями компьютеру совершить какое-либо действие. В этом отношении операторы C++ похожи на операторы других языков.

Окончание оператора обозначается знаком «точка с запятой». Этот знак является обязательным, однако чаще всего забывается в процессе программирования. В некоторых языках, например в BASIC, признаком конца оператора является конец строки, но в отношении C++ это неверно. Если вы забудете поставить точку с запятой в конце оператора, то в большинстве случаев (хотя и не всегда) компилятор возвратит ошибку.

Второй оператор нашей программы return 0; является указанием функции main() возвратить значение 0 вызывающему окружению; в данном случае это может быть компилятор или операционная система. В более старых версиях C++ для функции main() можно было указывать тип возвращаемого значения void, но стандартный C++ не рекомендует этого делать. Оператор return будет более подробно рассмотрен в главе 5.

Разделяющие знаки

Мы уже говорили о том, что символ конца строки не обрабатывается компилятором C++. На самом деле компилятор игнорирует практически все разделяющие

знаки. К разделяющим знакам относятся пробелы, символы возврата каретки и перехода на другую строку, вертикальная и горизонтальная табуляция и перевод страницы. Эти символы не обрабатываются компилятором. Вы можете записать несколько операторов на одной строке, разделить их любым количеством пробелов, табуляций или пустых строк, и компилятор во всех случаях обработает их одинаково. Таким образом, наша программа FIRST может быть записана и так:

```
#include <iostream>
using
namespace std;

int main() { cout
<<
"У каждой эпохи свой язык\n"
; return
0; }
```

Мы не рекомендуем использовать такой стиль — он не является стандартизованным и неудобен для чтения, хотя и компилируется правильно.

Есть несколько исключений из общего правила, когда компилятор обрабатывает разделяющие символы. Первое исключение — строка #include программного кода, являющаяся директивой препроцессора и записывающаяся в одну строку. Строковые константы, такие, как "У каждой эпохи свой язык\n", нельзя разбивать на несколько строк кода. Если вам необходимо использовать длинную строковую константу, вы можете вставить символ обратной косой черты \ в место разбиения строки или разделить вашу строку на несколько более коротких подстрок, каждая из которых будет заключена в кавычки.

Вывод с использованием cout

Как мы уже видели, оператор

```
cout << "У каждой эпохи свой язык\n";
```

выводит на экран строку, заключенную в кавычки. Каким образом работает этот оператор? Чтобы полностью это понять, необходимо знать объекты, перегрузку операций и другие аспекты, рассматриваемые позже. Пока мы ограничимся лишь краткими пояснениями.

Идентификатор cout на самом деле является объектом C++, предназначенным для работы со *стандартным потоком вывода. Поток* — это некоторая абстракция, отражающая перемещение данных от источника к приемнику. Стандартный поток вывода обычно направлен на экран, хотя допускается возможность его перенаправления на другие устройства вывода. Мы рассмотрим потоки и их перенаправление в главе 12 «Потоки и файлы».

Операция << называется операцией *вставки*. Она копирует содержимое переменной, стоящей в правой ее части, в объект, содержащийся в левой ее части. В программе FIRST операция << направляет строку "У каждой эпохи свой язык\n" в переменную cout, которая выводит эту строку на экран.

Если вы знакомы с С, то операция << известна вам как операция *побитового сдвига влево*, и вы, наверное, недоумеваете, каким образом ее можно использовать в качестве средства ввода/вывода. В С++ операции могут быть перегружены, то есть могут выполнять различные действия в зависимости от контекста, в котором они встречаются. Мы научимся перегружать операции в главе 8 «Перегрузка операций».

Несмотря на то, что на данный момент использование cout и << может показаться не вполне ясным, оно на самом деле не представляет трудности, и будет использоваться почти во всех программах. Иллюстрация механизма действия cout и << приведена на рис. 2.2.

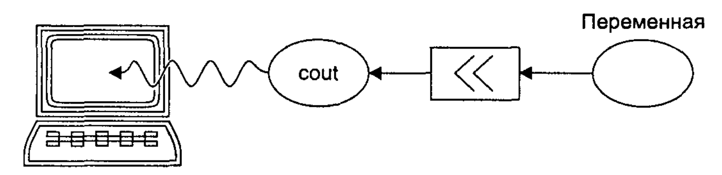


Рис. 2.2. Вывод с помощью cout

Строковые константы

Фраза "У каждой эпохи свой язык\n" является примером строковой константы. Как вы, возможно, знаете, константе, в отличие от выражения, нельзя придавать новое значение в процессе выполнения программы. Значение константы задается один раз и сохраняется на протяжении всего процесса выполнения программы.

Как мы позже увидим, работа со строками в С++ весьма запутанна. Как правило, применяются два способа интерпретации строк. В одном случае строка представляется как массив символов, в другом — как объект класса. Мы подробно рассмотрим оба типа строк в главе 7 «Массивы и строки».

Символ '\n' в конце строковой константы является примером *управляющей* или *escape-последовательности*. В данном случае такая последовательность означает, что следующий вывод текста начнется с новой строки. Это нужно, например, для того, чтобы фраза Press any key to continue или аналогичная, вставляемая большинством компиляторов после завершения программы, выводилась на новой строке. Мы еще вернемся к рассмотрению управляющих последовательностей в этой главе.

Директивы

Первые две строки, с которых начинается программа FIRST, являются *директивами*. Первая строка представляет собой *директиву препроцессора*, вторая — директиву using. Директивы нельзя считать частью языка С++, однако их использование является необходимым.

Директивы препроцессора

Первая строка программы FIRST

```
#include <iostream>
```

может показаться похожей на оператор, но это на самом деле не так. Она не входит в тело какой-либо из функций и не заканчивается точкой с запятой (;), как операторы C++. Кроме того, эта строка начинается с символа #. Такая строка называется *директивой препроцессора*. Вспомните, что любой оператор является указанием компьютеру совершить какое-либо действие, например сложить два числа или вывести на печать предложение. Директива препроцессора, напротив, является указанием компилятору. Препроцессором называется специальная часть компилятора, обрабатывающая подобные директивы перед началом процесса компиляции кода.

Директива #include указывает препроцессору включить в компилируемый файл содержимое другого файла. Другими словами, это содержимое подставляется на место директивы #include. Директива #include действует так же, как действуете вы, копируя необходимый текст в текстовом редакторе, а затем вставляя его в нужное место вашего файла.

#include является одной из многих директив препроцессора, каждая из которых предваряется символом #. Использование подобных директив в C++ не столь легко, как это было в C, но, тем не менее, мы рассмотрим несколько примеров.

Файл, включаемый с помощью директивы #include, обычно называют *заголовочным файлом*.

Заголовочные файлы

В программе FIRST директива #include является указанием включить в исходный текст содержимое файла IOSTREAM перед компиляцией кода. IOSTREAM является примером заголовочного (или включаемого) файла. Файл IOSTREAM содержит описания, необходимые для работы с переменной cout и операцией <<. Без этих описаний компилятору не будет известно, что значит имя cout, а употребление операции << будет воспринято как некорректное. Существует множество заголовочных файлов. В стандартном C++ заголовочные файлы не имеют расширения, но те файлы, которые были унаследованы от языка C, имеют расширение .H.

Если вам интересно заглянуть в содержимое файла IOSTREAM, то вы можете найти его в подкаталоге INCLUDE вашего компилятора и просмотреть его так же, как и любой исходный файл (указания, каким образом это сделать, можно найти в соответствующем приложении). Можно просмотреть заголовочный файл с помощью текстовых редакторов WordPad или Notepad. Содержимое заголовочного файла вряд ли окажется вам полезным, но, по крайней мере, вы убедитесь в том, что заголовочный файл выглядит как обычный исходный текстовый файл.

Мы еще вернемся к рассмотрению заголовочных файлов в конце этой главы, когда будем рассматривать библиотечные функции.

Директива using

Каждую программу на языке C++ можно разбить на несколько так называемых *пространств имен. Пространством имен* называется область программы, в которой распознается определенная совокупность имен. Эти имена неизвестны за пределами данного пространства имен. Директива

```
using namespace std;
```

означает, что все определенные ниже имена в программе будут относиться к пространству имен с именем std. Различные элементы программы описаны с использованием пространства имен std, например переменная cout. Если не использовать директиву using, то к этим элементам программы придется каждый раз добавлять имя std:

```
std::cout << "У каждой эпохи свой язык\n";
```

Для того чтобы не дописывать std:: каждый раз перед именем переменной, используется директива using. Подробнее пространства имен мы обсудим в главе 13 «Многофайловые программы».

Комментарии

Комментарии являются важной частью любой программы. Они помогают разобраться в действиях программы как разработчику, так и любому другому человеку, читающему код. Компилятор игнорирует все, что помечено в программе как комментарий, поэтому комментарии не включаются в содержимое исполняемого файла и никак не влияют на ход исполнения программы.

Синтаксис комментариев

Давайте перепишем программу FIRST, дополнив ее комментариями, и назовем получившийся пример COMMENTS:

```
//comments.cpp
//демонстрирует использование комментариев
#include <iostream>         //директива препроцессора
using namespace std;        //директива using
int main()                  //функция с именем main
{                           //начало тела функции
  cout << "У каждой эпохи свой язык\n"; //оператор
  return 0;                 //оператор
}                           //конец тела функции
```

Комментарии начинаются с двойной косой черты // и заканчиваются концом строки (этот случай как раз является примером, когда компилятор не игнорирует разделяющие символы). Комментарий может начинаться как в начале строки, так и после какого-либо оператора. В программе COMMENTS используются оба варианта комментария.

Использование комментариев

Комментарии полезны почти всегда, хотя многие программисты не используют их в полном объеме. Если вы решили не прибегать к помощи комментариев, то стоит помнить о том, что не каждый человек может разобраться в вашем коде так же легко, как это удается вам; возможно, что ваши действия требуют дополнительного разъяснения. Кроме того, по прошествии некоторого времени вы сами можете забыть некоторые детали алгоритма своей программы.

Используйте комментарии так, чтобы человек, читающий листинг вашей программы, понимáл, что делает эта программа. Детали алгоритма можно понять из самих операторов, поэтому задача комментариев — дать общие пояснения функционирования отдельных блоков программы.

Альтернативный вид комментариев

В C++ существует еще один вид комментариев:

```
/* устаревший вид комментариев */
```

Этот вид комментариев — единственный, применяемый в языке С. Признаком его начала служит последовательность символов /*, а заканчивается комментарий последовательностью */ (а не символом конца строки). Последовательности /* и */ труднее набирать, поскольку / относится к нижнему регистру, а * — к верхнему. Кроме того, неудобно каждый раз набирать два разных символа. *Этот стиль обычно не используется в C++,* но у него есть преимущество: если необходимо написать многострочный комментарий, то потребуется всего лишь два символа начала комментария:

```
/* Это
длинный
многострочный
комментарий
*/
```

В случае первого типа комментария пришлось бы в начало каждой строки вставлять символы //.

Комментарий в стиле /* */ можно вставлять в любое место строки:

```
func1()
  { /* пустое тело функции */ }
```

Если в данном случае заменить /* на //, то закрывающая скобка }, обозначающая конец тела функции, в процессе компиляции будет проигнорирована компилятором и вызовет ошибку.

Переменные целого типа

Переменные являются фундаментальной частью любого языка. Каждая переменная имеет символьное имя и может принимать различные значения. Переменные хранятся в определенных участках памяти компьютера. Когда переменной при-

сваивается значение, оно записывается в ячейку памяти, связанную с данной переменной. Большинство из распространенных языков программирования поддерживают определенный набор типов переменных, в который входят целые, вещественные и символьные типы.

Целые типы соответствуют целым числам, таким, как 1, 30 000, −27. Этот тип данных нужен для того, чтобы оперировать с дискретным числом объектов. В отличие от вещественных чисел, целые числа лишены дробной части.

Описание переменных целого типа

Существует несколько целых типов данных, имеющих различную длину, однако самым распространенным среди них является тип int. Размер переменной, имеющей такой тип, является системно-зависимым. В 32-разрядных операционных системах, таких, как Windows, размер переменной int равен 4 байтам (32 битам), что позволяет хранить в переменной типа int значения от −2 147 483 648 до 2 147 483 647. На рис. 2.3 изображено расположение целой переменной в памяти.

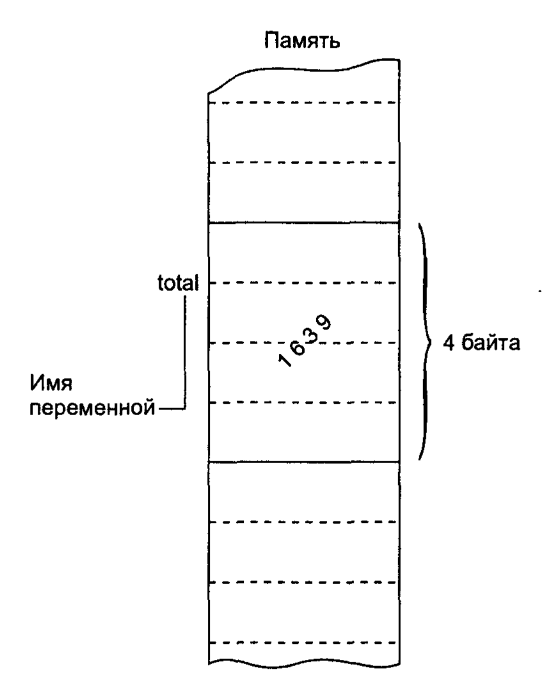


Рис. 2.3. Расположение переменной типа int в памяти

В MS DOS и ранних версиях Windows тип int занимал всего 2 байта. Сведения о диапазонах значений переменных стандартных типов хранятся в заголовочном файле LIMITS; информацию о них также можно получить в справочной системе вашего компилятора.

Приведем пример программы, работающей с переменными типа int:

```
//intvars.cpp
// работа с переменными целого типа
#include <iostream>
using namespace std;
int main()
{
  int var1;                      // описание переменной var1
  int var2;                      // описание переменной var2
  var1 = 20;                     // присвоение значения переменной var1
  var2 = var1 + 10;              // присвоение значения переменной var2
  cout << "var1+10 равно ";      // вывод строки
  cout << var2 << endl;          // вывод значения переменной var2
  return 0;
}
```

Откройте текст этой программы в окне редактирования вашего компилятора, затем скомпилируйте и запустите программу и просмотрите содержимое окна вывода. Операторы

```
int var1;
int var2;
```

определяют две целые переменные var1 и var2. Ключевое слово int определяет тип переменной. Подобные операторы, называемые *объявлениями*, должны разделяться точкой с запятой (;), как и любые другие операторы программы.

Перед тем как воспользоваться переменной, вы обязательно должны объявить ее. Объявления переменных могут располагаться в любом месте программы. Не обязательно делать все объявления переменных до появления первого исполняемого оператора, как это было принято в С. Тем не менее, объявления наиболее часто употребляемых переменных целесообразно производить в начале программы, чтобы обеспечить ее удобочитаемость.

Объявление и определение переменной

Давайте немного вернемся назад и поясним разницу между терминами *определение переменной* и *объявление переменной.*

Объявление переменной предполагает указание имени переменной (например, var1) и ее типа (int и т. д.). Если при объявлении переменной одновременно выделяется память под нее, то происходит *определение переменной.* Операторы

```
int var1;
int var2;
```

в программе INTVARS являются и объявлениями, и определениями, поскольку они выделяют память под переменные var1 и var2. Мы будем чаще употреблять те объявления, которые являются определениями, но позже мы познакомимся и с теми видами объявлений, которые не являются определениями.

Имена переменных

В программе INTVARS используются переменные с именами var1 и var2. Имена, даваемые переменным и другим элементам программы, называются *идентификаторами*. Каковы правила составления идентификатора? Вы можете использовать буквы как верхнего, так и нижнего регистров, а также цифры от 1 до 9. Кроме того, разрешается использовать символ подчеркивания _. Первый символ идентификатора должен быть либо буквой, либо символом подчеркивания. Длина идентификатора теоретически не ограничивается, но большинство компиляторов не распознают идентификаторы длиной более нескольких сотен символов. Компилятор различает буквы верхнего и нижнего регистра, поэтому имена Var и VAR будут восприниматься как различные.

В качестве имен переменных в C++ нельзя использовать ключевые слова. Ключевое слово — это зарезервированное слово, имеющее особое значение. Примерами ключевых слов могут служить int, class, if, while. Полный список ключевых слов можно найти в приложении Б или в справочной системе вашего компилятора.

Многие программисты на C++ придерживаются негласного правила использовать в названиях переменных буквы только нижнего регистра. Есть программисты, которые употребляют в именах переменных как строчные, так и прописные буквы: IntVar или dataCount. Некоторые предпочитают прибегать к помощи символа подчеркивания. Какой бы подход вы ни использовали, желательно последовательно придерживаться его в рамках программы. Имена, состоящие только из букв верхнего регистра, иногда применяются для обозначения констант (разговор о константах пойдет чуть позже). Подобных соглашений, как правило, придерживаются и при составлении имен классов, функций и других элементов программы.

Желательно, чтобы имя переменной отражало смысл ее содержимого. Например, имя boilerTemperature является более предпочтительным, чем bT или t.

Операция присваивания

Операторы

```
var1 = 20;
var2 = var1 + 10;
```

присваивают значения переменным var1 и var2. Как вы, возможно, догадались, знак = означает присваивание значения, стоящего в правой его части, переменной, стоящей в левой части. Операция = является эквивалентом аналогичной операции языка BASIC и операции := языка Pascal. В первой строке приведенного здесь фрагмента переменной var1, до этого не имевшей значения, присваивается значение 20.

Целые константы

Число 20 является примером константы целого типа. Константы сохраняют свое значение на протяжении всего выполнения программы. Константа целого типа

может содержать только цифры. Использование десятичной точки в целой константе не допускается. Значение целой константы должно содержаться в интервале, соответствующем диапазону представления целых чисел.

Во второй строчке последнего фрагмента знак + означает сложение значения переменной var1 и числа 10, являющегося константой. Результат сложения присваивается переменной var2.

Оператор вывода

Оператор

```
cout << "var1+10 равно ";
```

выводит на экран строковую константу, о чем мы говорили раньше. Оператор

```
cout << var2 << endl;
```

отображает значение переменной var2. Как вы можете убедиться, взглянув на содержимое окна консоли, вывод нашей программы выглядит следующим образом:

```
var1+10 равно 30
```

Заметьте, что операция << и объект cout знают, каким образом отличать целое число от строки и как обрабатывать каждое из них. Если мы выводим на печать целое число, то оно отображается в числовом формате. Если мы печатаем строку, то она выводится в виде текста. Это кажется очевидным, но за этими действиями стоит механизм перегрузки операций, типичный для C++ (программисты на языке C знают, что для вывода переменной необходимо сообщить функции printf не только имя этой переменной, но и ее тип, что усложняет синтаксис команды).

Как видите, вывод, сделанный парой операторов cout, располагается в одной строке. Автоматического перехода на следующую строку нет, и если необходимо сделать такой переход, то приходится вставлять символ конца строки вручную. Мы уже рассмотрели применение управляющей последовательности \n; теперь мы рассмотрим другой способ перехода на следующую строку, реализуемый с помощью манипулятора.

Манипулятор endl

Второй оператор cout в программе INTVARS заканчивается неизвестным для нас словом endl. Это слово означает вставку в символьный поток символа окончания строки, поэтому весь последующий текст будет печататься с новой строки. Фактически это эквивалентно действию управляющей последовательности \n, но первый способ предпочтительнее. endl представляет собой манипулятор — особую инструкцию, обращенную к потоку и предназначенную для изменения вывода. Мы будем использовать в наших программах и другие манипуляторы. Строго говоря, манипулятор endl связан с очисткой выходного буфера, однако, как прави-

ло, это не имеет большого значения, и в большинстве случаев можно считать \n и endl эквивалентными.

Другие целые типы

Кроме int существует также еще несколько целых типов, из которых наиболее употребительными являются long и short (строго говоря, тип char тоже является целым, но пока мы не будем рассматривать его в таком качестве). Мы уже говорили, что размер переменных типа int является аппаратно-зависимым. Переменные типов long и short, напротив, имеют фиксированный размер, не зависящий от используемой системы.

Размер типа long всегда равен 4 байтам и совпадает с размером типа int в случае 32-разрядных систем, подобных Windows. Это означает, что диапазон значений типа long совпадает с диапазоном типа int: от −2 147 483 648 до 2 147 483 647. Тип long может быть описан как long int: между двумя такими описаниями нет разницы. Если ваша операционная система 32-разрядная, то не важно, какой тип использовать — int или long, но в 16-разрядной системе тип long сохранит свой диапазон значений, в то время как тип int уже будет иметь диапазон, совпадающий с типом short.

Тип short в любой операционной системе имеет размер, равный двум байтам. Диапазон значений типа short — от −32 768 до 32 767. Использование типа short не имеет особого смысла на современных 32-разрядных операционных системах, за исключением тех случаев, когда необходима экономия памяти. Несмотря на вдвое больший размер по сравнению с типом short, обработка переменных типа int происходит быстрее.

Если вам необходимо описать константу типа long, то после ее числового значения следует указать символ L:

```
longvar = 7678L; // описание константы longvar типа long
```

Многие компиляторы позволяют определять целые типы с указанием нужной разрядности (в битах). Имена таких типов начинаются с двойного символа подчеркивания: __int8, __int16, __int32, __int64. Тип __int8 соответствует типу char, типы __int16 и __int32 — соответственно типу short и паре типов int и long (справедливо как минимум для 32-разрядных систем). Тип __int64 используется для хранения больших целых чисел разрядностью до 19 десятичных знаков. Преимущество этих типов данных состоит в том, что они не зависят от операционной системы, в которой они используются, но, тем не менее, использование этих типов не получило большого распространения.

Символьные переменные

Символьные переменные хранят целые числа, содержащиеся в диапазоне от −128 до 127. Размер памяти, занимаемый такими переменными, равен 1 байту (8 битам). Иногда символьные переменные используют для представления целых чисел,

заключенных в указанном диапазоне, но гораздо чаще в таких переменных хранятся ASCII-коды символов.

Как вы уже, возможно, знаете, таблица ASCII-кодов предназначена для интерпретации символов как чисел. Эти числа заключены в диапазоне от 0 до 127. Большинство операционных систем семейства Windows расширяют верхнюю границу этого диапазона до 255 для того, чтобы включить в ASCII-таблицу символы национальных алфавитов и псевдографические символы. В приложении А приведена таблица ASCII-символов.

Из-за того, что не существует стандарта для символов, соответствующих числам от 128 до 255, а также потому, что 256 чисел недостаточно для представления символов всех существующих алфавитов: при потребности в иностранных символах могут возникать проблемы. Проблемы могут возникать даже при работе одной программы на двух разных компьютерах, использующих один и тот же язык, совпадающий с тем, который реализован в программе. Чтобы разрешить проблему использования символов из различных алфавитов, в стандартном C++ применяется расширенный символьный тип wchar_t. Разумеется, это оказывается полезным при написании программ, предназначенных для использования в различных странах мира. Однако в этой книге мы не будем касаться работы с типом wchar_t и ограничимся рассмотрением только символов таблицы ASCII операционной системы Windows.

Символьные константы

Символьные константы записываются в одиночных кавычках: 'a', 'b' и т. д. (обратите внимание на то, что символьные константы записываются в одиночных кавычках, в то время как строковые константы — в двойных). Когда компилятор встречает символьную константу, он заменяет ее на соответствующий ASCII-код. Например, константа 'a' будет заменена числом 97, как показано на рис. 2.4.

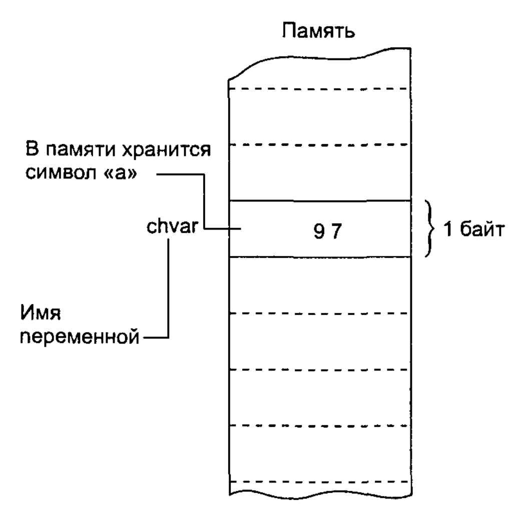


Рис. 2.4. Расположение переменной типа char в памяти

Символьным переменным можно присваивать значения символьных констант. Пример работы с символьными переменными и константами приведен ниже.

```
// charvars.cpp
// применение символьных констант
#include <iostream>          // для cout и т.п.
using namespace std;
int main()
{
  char charvar1 ='A ';       // символьная переменная
                             // со значением 'A'
  char charvar2 ='\t ';      // символьная переменная со значением символа табуляции
  cout <<charvar1;           // вывод переменной на экран
  cout <<charvar2;           // вывод переменной на экран
  charvar1 ='B';             // присваивание константного
                             // значения символьной
                             // переменной
  cout <<charvar1;           // вывод переменной на экран
  cout <<'\n ';              // переход на следующую строку
  return 0;
}
```

Инициализация

Инициализация переменных возможна одновременно с их объявлением. В приведенной выше программе переменные charvar1 и charvar2 типа char инициализируются константными значениями 'A' и '\t'.

Управляющие последовательности

В качестве примеров управляющих последовательностей можно привести '\n', уже упоминавшуюся в этой главе, и '\t', используемую в последнем примере. Название «управляющая последовательность» означает, что символ \ «управляет» интерпретацией следующих за ним символов последовательности. Так, t воспринимается не как символ 't', а как символ табуляции. Символ табуляции означает, что весь поток вывода будет условно разделен на фрагменты одинаковой длины, определяемой шагом табуляции, и следующий символ будет напечатан в начале следующего фрагмента, а не сразу за предыдущим символом. В консольных программах шаг табуляции равен восьми позициям. Символьная константа '\n' посылается объекту cout в последней строке программы.

Управляющие последовательности можно использовать как в качестве отдельных констант, так и в составе строковых констант. Список управляющих последовательностей приведен в табл. 2.1.

Поскольку при употреблении символьных и строковых констант символ \, а также одинарные и двойные кавычки по-особому интерпретируются компилятором, необходимо с помощью управляющих последовательностей обеспечить способ их включения в символьные и строковые константы в качестве обычных символов. Вот пример реализации вывода на экран строки с кавычками:

```
cout << "\"Ну все, мы полетели\", сказала она.";
```

Выводимая на экран строка будет иметь вид

```
"Ну все, мы полетели", сказала она.
```

Таблица 2.1. Управляющие последовательности

Управляющая последовательность	Символ
\a	Сигнал
\b	Пробел
\f	Перевод страницы
\n	Перевод в начало следующей строки
\r	Возврат каретки
\t	Табуляция
\\	Обратная косая черта
\'	Одинарные кавычки
\"	Двойные кавычки
\xdd	Шестнадцатеричный код символа

Иногда бывает необходимо описать символьную константу, которая не может быть введена с клавиатуры, например, псевдографический символ с ASCII-кодом 127. Чтобы сделать это, можно использовать управляющую последовательность вида '\xdd', где d обозначает шестнадцатеричную цифру. Если вам захочется напечатать символ, представляющий собой закрашенный прямоугольник, то вы должны найти в ASCII-таблице код этого символа — 178. Если перевести число 178 в шестнадцатеричную систему счисления, то получим число B2. Таким образом, нужный символ представляется управляющей последовательностью '\xB2'. Мы приведем еще несколько подобных примеров позже.

Программа CHARVARS печатает на экране значения переменных charvar1 ('A') и charvar2 (символ табуляции). Затем программа меняет значение переменной charvar1 на 'B', печатает его, и в конце выводит символ перехода на другую строку. Результат работы программы выглядит следующим образом:

```
A    B
```

Ввод с помощью cin

Теперь, когда мы познакомились с основными типами данных и поработали с ними, рассмотрим, каким образом программа осуществляет ввод данных. Следующая программа просит пользователя ввести значение температуры по Фаренгейту, затем переводит это значение в шкалу Цельсия и отображает результат на экране. В программе используются переменные целого типа.

```
//fahren.cpp
//применение cin и \n
#include <iostream>
using namespace std;
int main()
{
```

```
    int ftemp;                  // температура по Фаренгейту
    cout << "Введите температуру по Фаренгейту: ";
    cin >> ftemp;
    int ctemp = (ftemp-32)*5 / 9;
    cout << "Температура по Цельсию равна " << ctemp << '\n ';
    return 0;
}
```

Оператор

```
cin >> ftemp;
```

заставляет программу ожидать ввода числа от пользователя. Введенное значение присваивается переменной ftemp. Ключевое слово cin является объектом, определенным в C++ для работы со стандартным потоком ввода. Этот поток содержит данные, вводимые с клавиатуры (если он не переопределен). >> является *операцией извлечения*. Она извлекает данные из потокового объекта, стоящего в левой части, и присваивает эти данные переменной, стоящей в правой части.

Результат работы программы может быть следующим:

```
Введите температуру по Фаренгейту: 212
Температура по Цельсию равна 100
```

На рис. 2.5 изображена схема функционирования cin и >>.

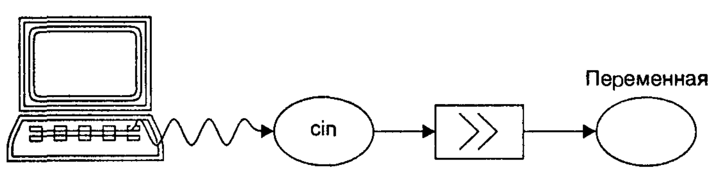


Рис. 2.5. Ввод с помощью cin

Определение переменных при первом использовании

Программа FAHREN использует, помимо ввода информации, несколько новых для нас средств. Давайте внимательно взглянем на листинг. Где определена переменная ctemp? В начале программы ее определение отсутствует. Определение переменной ctemp находится в предпоследней строке, где она используется для сохранения результата арифметической операции. Как мы уже говорили, вы можете определять переменную в любом месте программы, и совсем не обязательно делать это перед первым исполняемым оператором, как в языке С.

Определение переменных там, где они используются, делает листинг программы проще для понимания, поскольку нет необходимости постоянно возвращаться в начало программы для того, чтобы найти описания нужных переменных. Однако необходимо благоразумно подходить к применению этого метода: если переменные многократно используются в разных частях функции, то во избежание путаницы лучше определять их в начале функции.

Каскадирование операции <<

В рамках второго из операторов cout в программе fahren операция << повторяется несколько раз, или *каскадируется*. Такая конструкция является вполне законной, потому что в этом случае операция << сначала посылает в переменную cout строку "Температура по Цельсию равна ", затем значение переменной ctemp, и наконец, символ перехода на новую строку '\n'.

Операцию извлечения >> можно каскадировать совместно с cin аналогичным путем, давая возможность пользователю вводить несколько значений подряд. Однако такой подход употребляется достаточно редко, поскольку в этом случае перед вводом значений на экран пользователю не будут выводиться соответствующие приглашения.

Выражения

Любая комбинация переменных, констант и операций, приводящая к вычислению некоторого значения, называется *выражением*. Например, выражениями являются конструкции alpha+12 или (alpha-37)*beta/2. Результатом выполнения всех операций, входящих в состав выражения, является значение. Так, например, если alpha будет равно 7, то значение первого выражения будет равно 19.

Выражения сами могут входить в состав других выражений. Во втором примере alpha-37 и beta/2 являются выражениями. Фактически каждую переменную и константу, например alpha и 37, можно считать частным случаем выражения.

Обратите внимание, что выражения и операторы — это не одно и то же. Операторы являются указанием компьютеру совершить какое-либо действие и всегда завершаются точкой с запятой (;). Выражения же лишь определяют некоторую совокупность вычислений. В одном операторе могут присутствовать несколько выражений.

Приоритеты выполнения операций

Обратите внимание на скобки, присутствующие в выражении

```
(ftemp-32)*5 / 9
```

Если убрать эти скобки, то операция умножения будет выполнена первой, поскольку обладает более высоким приоритетом, чем операция вычитания. Скобки заставляют операцию вычитания выполняться перед операцией умножения, поскольку операции, заключенные в скобки, выполняются раньше. А как обстоит дело с отношением приоритетов операций умножения и деления? Если приоритеты двух операций равны, то первой из них будет выполнена та, которая стоит слева. В нашем примере сначала будет выполнена операция умножения, а затем — деления.

Такое использование приоритетов аналогично алгебре и другим языкам программирования. Это кажется вполне естественным на интуитивном уровне, одна-

ко раздел, касающийся приоритетов операций в C++, является важным для изучения, и мы еще вернемся к нему после того, как рассмотрим другие операции языка.

Вещественные типы

Мы уже рассматривали такие типы данных, как int и char, которые представляют целые числа, то есть числа, у которых нет дробной части. Теперь мы рассмотрим типы данных, позволяющие хранить другой класс чисел — вещественный.

Переменные вещественного типа хранят числа в десятичной форме представления, например 3.1415927, 0.0000625, –10.2. У таких чисел есть как целая часть, стоящая слева от десятичной точки, так и дробная часть, стоящая справа от нее. Переменные вещественного типа предназначены для хранения вещественных чисел — тех чисел, которыми измеряются непрерывные величины: температура, расстояние, площадь. Вещественные числа, как правило, имеют ненулевую дробную часть.

В C++ имеются три вещественных типа: float, double и long double. Рассмотрим первый из них.

Тип float

Тип float способен хранить числа, содержащиеся в интервале от $3.4*10^{-38}$ до $3.4*10^{38}$, с точностью до семи знаков после запятой. Размер типа float равен 4 байтам (32 битам), как показано на рис. 2.6.

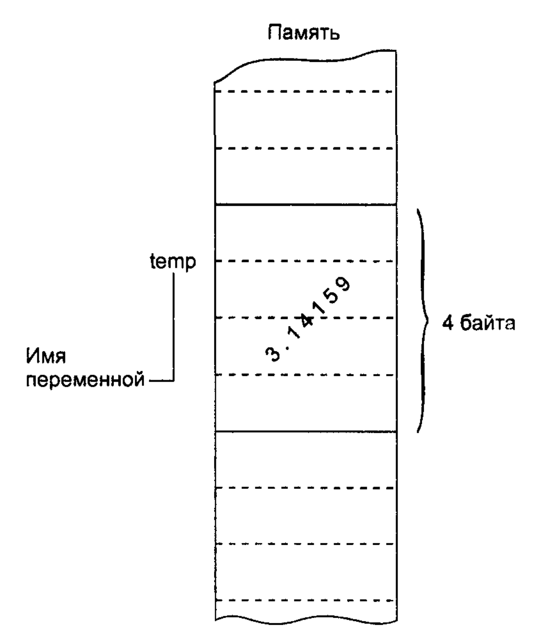


Рис. 2.6. Расположение переменной типа float в памяти

Следующий пример демонстрирует работу с вещественными числами. Пользователю предлагается ввести радиус окружности, а программа подсчитывает и выводит на экран площадь круга, ограниченного данной окружностью.

```
//circarea.cpp
//работа с переменными вещественного типа
#include <iostream>          //для cout и т.д.
using namespace std:
int main()
{
  float rad:                  // переменная вещественного типа
  const float PI =3.14159F:   // вещественная константа
  cout << "Введите радиус окружности: ": // запрос
  cin >> rad:                 // получение радиуса
  float area = PI *rad *rad: // вычисление площади круга
  cout <<"Площадь круга равна " << area << endl: // вывод результата на экран
  return 0:
}
```

Примером результата работы такой программы может служить следующий:

```
Введите радиус окружности: 0.5
Площадь круга равна 0.785398
```

Здесь мы подсчитали площадь 12-дюймовой долгоиграющей грампластинки, выраженную в квадратных футах. В свое время эта величина имела важное значение для производителей винила.

Типы double и long double

Два более крупных вещественных типа данных — double и long double — аналогичны типу float и отличаются от него лишь размерами занимаемой памяти, диапазонами значений и точностью представления. Тип double занимает 8 байтов памяти и хранит значения от $1.7*10^{-308}$ до $1.7*10^{308}$ с точностью до 15 знаков после запятой. Характеристики типа long double зависят от компилятора, но чаще всего они совпадают с характеристиками типа double. Иллюстрация типа double приведена на рис. 2.7.

Вещественные константы

Число 3.14159F в программе CIRCAREA является примером вещественной константы. Наличие десятичной точки говорит о том, что тип этой константы вещественный, а не целый, а суффикс F после значения константы указывает конкретный ее тип — float. Форма представления значения константы — нормализованная десятичная. Если вы определяете вещественную константу типа double, то суффикс D не обязателен — он является суффиксом по умолчанию. Для констант типа long double необходимо указывать суффикс L.

Для задания значений вещественных констант наряду с десятичной может также использоваться экспоненциальная форма записи. Экспоненциальная форма записи предназначена для представления очень больших или очень малень-

ких чисел, чтобы не выписывать большое количество нулей. Например, число 1 000 000 000 можно записать в виде 1.0E-9. Аналогично, число 1234.56 будет представлено в экспоненциальной форме как 1.23456E3, что соответствует арифметической форме записи $1.23456*10^3$. Число, следующее за знаком E, называется экспонентой числа. Экспонента числа показывает, на сколько позиций необходимо переместить десятичную точку для того, чтобы вернуть число к нормальной десятичной форме.

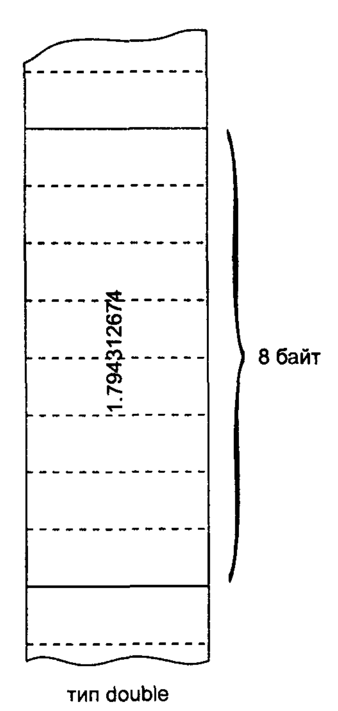


Рис. 2.7. Расположение переменной типа double в памяти

Экспонента числа может быть как положительной, так и отрицательной. Экспоненциальная форма 6.35239E-5 соответствует десятичной форме 0.0000635239, что совпадает с числом $6.35239*10^{-5}$.

Префикс const

Кроме работы с переменными типа float, программа CIRCAREA демонстрирует использование префикса const. Он используется в операторе

```
const float PI = 3.14159F;   //тип вещественная константа
```

Ключевое слово const предшествует описанию типа переменной и означает, что во время выполнения программы запрещено изменять значение этой переменной. Любая попытка изменить значение переменной, описанной с таким префиксом, приведет к выдаче компилятором сообщения об ошибке.

Префикс const гарантирует, что наша программа не сможет случайно изменить значение переменной. Примером может служить переменная PI в программе CIRCAREA. Кроме того, префикс const информирует читающего листинг о том, что значение переменной не будет изменяться в ходе выполнения программы. Как мы увидим позже, префикс const может применяться не только к переменным, но и к другим элементам программы.

Директива #define

Константы можно определять с помощью директивы препроцессора #define, несмотря на то, что такой способ не рекомендуется употреблять в C++. Директива #define не делает различий между числами и символьными строками. Например, если в начале вашей программы указана строчка

```
#define PI 3.14159
```

то идентификатор PI при компиляции будет заменен текстом 3.14159 везде, где он встречается. Такая конструкция долгое время была популярна в языке C. Однако отсутствие типа у подобных переменных может привести к некорректной работе программы, и поэтому даже в языке C был предложен способ, использующий слово const. В старых программах вы, тем не менее, можете найти конструкции с применением #define.

Тип bool

Чтобы завершить разговор о типах данных, мы должны упомянуть тип bool, хотя он не понадобится нам до тех пор, пока мы не рассмотрим операции отношения в следующей главе.

Мы видели, что у переменной типа int может быть несколько миллиардов различных значений; у типа char этих значений 256. Тип bool может иметь всего два значения — true и false. Теоретически размер переменной типа bool равен 1 биту (не байту!), но большинство компиляторов на практике выделяет под такие переменные 1 байт, поскольку доступ к целому байту осуществляется быстрее, чем к отдельному биту. Чтобы получить доступ к биту, необходимо произвести операцию его извлечения из того байта, в котором он содержится, что увеличивает время доступа.

Как мы увидим, переменные типа bool чаще всего используются для хранения результатов различных сравнений. Например, если значение переменной alpha меньше значения переменой beta, то переменной типа bool будет присвоено значение true, а в противном случае — false. Своим названием тип bool обязан фамилии английского математика XIX века Джорджа Булла, разработавшего концепцию применения логических операций с переменными типа «ложь—истина». Подобные переменные часто называют булевыми.

Манипулятор setw

Мы уже говорили о том, что манипуляторы — это особые операции, используемые совместно с операцией вставки << для того, чтобы видоизменять вывод, который делает программа. Мы познакомились с манипулятором endl, а теперь введем в рассмотрение еще один манипулятор — setw, который изменяет ширину поля вывода.

Можно рассматривать каждую переменную, выводимую с помощью объекта cout, как занимающую некоторое поле — воображаемое пространство с определенной длиной. По умолчанию такое поле имеет достаточный размер для того, чтобы хранить нужную переменную. Так, например, целое число 567 займет размер в 3 символа, а текстовая строка "pajamas" — 7 символов. Разумеется, существуют ситуации, когда подобный механизм не является удобным. Приведем пример программы WIDTH1, печатающей названия городов и численность их населения в виде двух столбцов:

```
//width1.cpp
//демонстрирует необходимость применения манипулятора setw
#include <iostream>
using namespace std;
int main()
{
  long pop1 = 4789426, pop2 = 274124, pop3 = 9761;
  cout << "Город    " << "Нас." << endl
  <<"Москва " << pop1 <<endl
  <<"Киров " << pop2 <<endl
  <<"Угрюмовка " <<pop3 <<endl;
  return 0;
}
```

Вывод программы будет выглядеть следующим образом:

```
Город  Нас.
Москва 4789426
Киров 274124
Угрюмовка 9761
```

Очевиден недостаток такого формата печати: очень неудобно визуально сравнивать числа между собой. Было бы гораздо приятней читать печатаемую информацию, если бы второй столбец был выровнен по правому краю. Но тогда нам пришлось бы вставлять в имена городов нужное количество пробелов, что также неудобно. Теперь мы приведем пример программы WIDTH2, решающей данную проблему путем использования манипулятора setw, который определяет длину полей имен городов и численности населения:

```
//width2.cpp
//применение манипулятора setw
#include <iostream>
#include <iomanip>        // для использования setw
using namespace std;
int main()
{
  long pop1 = 8425785, pop2 = 47, pop3 = 9761;
```

```
  cout << setw(9) << "Город " << setw(12)
  << "Население " << endl
  << setw(9) << "Москва" << setw(12) << pop1 << endl
  << setw(9) << "Киров" << setw(12) << pop2 << endl
  << setw(9) << "Угрюмовка" << setw(12) << pop3 << endl;
  return 0;
}
```

Манипулятор setw печатает число или строку, следующую за ним в потоке, в поле фиксированной длины n, где n — аргумент манипулятора setw(n). Значение переменной, помещаемой в поле, выравнивается справа. Рисунок 2.8 демонстрирует, как это делается. Для того чтобы избежать переполнения значения переменной, хранящей численность населения города, в низкоразрядных операционных системах для такой переменной определен тип long.

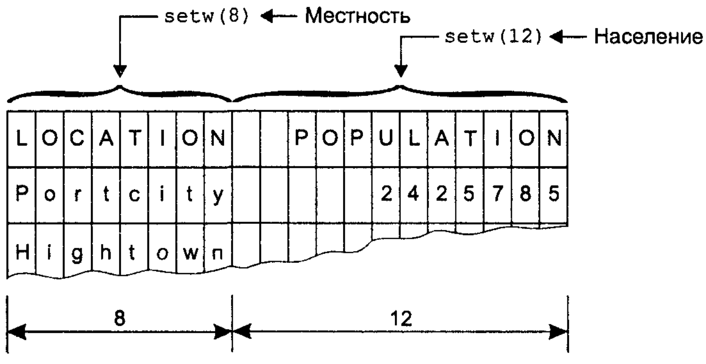


Рис. 2.8. Длина полей и setw

Вот результат работы программы WIDTH2:

```
ГородНаселение
Москва8425785
Киров47
Угрюмовка9761
```

Каскадирование операции <<

Обратите внимание на то, что в программах WIDTH1 и WIDTH2 имеется только один оператор с cout, несмотря на то, что он расположен на нескольких строчках. Записывая этот оператор подобным образом, мы прибегаем к свойству компилятора игнорировать разделительные символы. Таким образом, мы каскадируем операцию вставки, то есть записываем в одном операторе то, что можно было бы реализовать с помощью четырех последовательных операторов, начинающихся с cout.

Множественное определение

Мы инициализировали переменные pop1, pop2 и pop3 одновременно с их определением. Подобные действия мы производили с символьными переменными в

программе charvar. Однако разница заключается в том, что в первом случае мы сделали это с тремя переменными в одной строке, лишь один раз использовав ключевое слово long и разделив объявляемые переменные запятыми. Такой способ сокращает объем кода, поскольку объединяет объявление нескольких переменных одного типа.

Файл заголовка IOMANIP

Объявления манипуляторов (за исключением endl) происходит не в файле IOSTREAM, а в другом заголовочном файле — IOMANIP. Когда у вас возникнет необходимость использовать эти манипуляторы, вам придется включить этот файл с помощью директивы #include, подобно тому, как мы делали в программе WIDTH2.

Таблица типов переменных

До сих пор мы использовали в наших примерах переменные четырех типов: int, char, float и long. Кроме того, мы упоминали такие типы данных, как bool, short, double и long double. Давайте теперь подведем итог всему вышесказанному. В табл. 2.2 содержатся ключевые слова, описывающие типы данных, диапазоны значений и размер каждого типа в байтах в 32-разрядной операционной системе, а для вещественных типов данных также приведена точность представления данных в количестве знаков.

Таблица 2.2. Стандартные типы C++

Название типа	Нижняя граница диапазона	Верхняя граница диапазона	Точность	Размер в байтах
bool	False	True	Нет	1
char	−128	127	Нет	1
short	-32 768	32 767	Нет	2
int	-2 147 483 648	2 147 483 647	Нет	4
long	-2 147 483 648	2 147 483 647	Нет	4
float	$3.4*10^{-38}$	$3.4*10^{38}$	7	4
double	$1.7*10^{-308}$	$1.7*10^{308}$	15	8

Беззнаковые типы данных

Если исключить из представления целых чисел знак, то полученный тип данных будет представлять неотрицательные целые числа с удвоенной верхней границей диапазона представления. Беззнаковые типы данных перечислены в табл. 2.3.

Беззнаковые типы данных применяются в тех случаях, когда нет необходимости хранить в переменных отрицательные значения, например при подсчете количества чего-либо, или в тех случаях, когда верхняя граница диапазона обычного целого типа оказывается недостаточно велика.

Таблица 2.3. Беззнаковые целые типы

Названив	Нижняя граница диапазона	Вврхняя граница диапвзона	Размвр в байтах
unsigned char	0	255	1
unsigned short	0	65 535	2
unsigned int	0	4 294 967 295	4
unsigned long	0	4 294 967 295	4

Чтобы сделать целый тип беззнаковым, предварите его название ключевым словом unsigned. Например, определение беззнаковой переменной типа char будет выглядеть следующим образом:

```
unsigned char ucharv;
```

Выход за рамки допустимых для данного типа значений, как правило, приводит к труднообнаруживаемым ошибкам. Подобные ошибки с беззнаковыми типами происходят гораздо реже. Следующая программа хранит одно и то же значение, равное 1 500 000 000, в переменных signedVar типа int и unsignVar типа unsigned int.

```
// signtest.cpp
// работа со знаковыми / беззнаковыми переменными
#include <iostream>
using namespace std;
int main()
{
  int signedVar = 1500000000;            //знаковая переменная
  unsigned int unsignVar = 1500000000;   //беззнаковая переменная
  signedVar = (signedVar *2) / 3;        //выход за границы диапазона
  unsignVar = (unsignVar *2) / 3;        // вычисления внутри диапазона
  cout << "Знаковая переменная равна " << signedVar << endl;
                                         // вывод правильного результата
  cout << "Беззнаковая переменная равна " << unsignVar << endl;
                                         // вывод некорректного результата
  return 0;
}
```

Программа умножает обе переменные на 2, а затем делит на 3. Несмотря на то, что правильный результат должен получиться меньше исходного значения, промежуточное вычисление приводит к результату, большему, чем исходное число. Такая ситуация стандартна, но зачастую она приводит к ошибкам. Так, например, в программе SIGNTEST мы должны ожидать того, что в обеих переменных в итоге будет содержаться одно и то же значение, составляющее 2/3 от исходного и равное 1 000 000 000. Но результат умножения, равный 3 000 000 000, вышел за допустимый верхний предел для переменной signedVar, равный 2 147 483 647. Поэтому результат работы программы будет выглядеть следующим образом:

```
Знаковая переменная равна 431655765
Беззнаковая переменная равна 1000000000
```

Теперь в переменной со знаком содержится неверное значение, в то время как переменная без знака, которая имела достаточный диапазон для представления результата умножения, содержит корректный результат. Отсюда следует вывод: необходимо следить за тем, чтобы все значения, которые вычисляются в вашей программе, соответствовали диапазонам значений тех переменных, которым они присваиваются (при этом нужно учитывать, что на 16- и 64-битных компьютерах диапазоны даже для переменных типа int будут различными).

Преобразования типов

Язык C++, как и его предшественник C, свободнее, чем многие другие языки программирования, обращается с выражениями, включающими в себя различные типы данных. В качестве примера рассмотрим программу MIXED:

```
// mixed.cpp
// использование смешанных выражений
#include <iostream>
using namespace std;
int main()
{
  int count = 7;
  float avgWeight = 155.5F;
  double totalWeight = count *avgWeight;
  cout <<"Вес равен " << totalWeight << endl;
  return 0;
}
```

Здесь переменная типа int умножается на переменную типа float, а результат присваивается переменной типа double. Компиляция программы происходит без ошибок, поскольку компиляторы допускают возможность перемножения (и выполнения других арифметических операций) с операндами разных типов.

Не все языки поддерживают смешанные выражения, и некоторые из них фиксируют ошибки при попытке применить арифметические операции к данным разных типов. Такой подход предполагает, что подобные действия вызваны ошибкой программиста, и призван «сигнализировать» ему об этом. C и C++ предполагают, что смешивание типов данных произошло сознательно и является задумкой программиста, и поэтому не мешают ему реализовывать свои идеи. Подобный подход отчасти объясняет столь значительную популярность языков C и C++. Эти языки дают программисту большую свободу. Разумеется, подобная либеральность увеличивает вероятность допущения ошибок.

Неявные преобразования типов

Давайте рассмотрим действия компилятора, когда он встречает выражения со смешанными типами, подобные приведенным в программе MIXED. Каждый тип данных можно условно считать «ниже» или «выше» по отношению к другим типам. Иерархия типов данных приведена в табл. 2.4.

Таблица 2.4. Иерархия типов данных

Тип данных	Старшинство
long double	Высший
double	
float	
long	
int	
short	
char	Низший

Арифметические операции, подобные + и *, действуют следующим образом: если их операнды имеют различные типы, то операнд с более «низким» типом будет преобразован к более «высокому» типу. Так, в программе MIXED тип int переменной count был преобразован в float с помощью введения временной переменной, содержимое которой умножается на переменную avgWeight. Результат, имеющий тип float, затем преобразовывается к типу double, чтобы его можно было присвоить переменной totalWeight. Процесс преобразования типов показан на рис. 2.9.

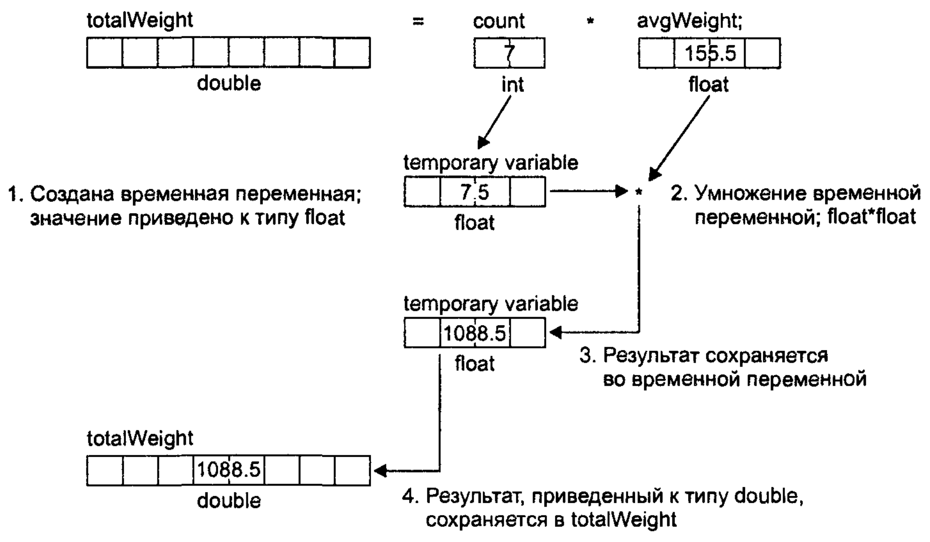


Рис. 2.9. Преобразования типов

Подобные преобразования типов данных происходят неявно, и обычно нет необходимости задумываться над ними, поскольку C++ сам выполняет то, что мы хотим. Однако в ряде случаев компилятор не столь понятно проводит преобразования типов, и мы скоро в этом убедимся. Когда мы научимся работать с объектами, мы фактически будем оперировать собственными типами данных. Возможно, нам захочется применять переменные таких типов в смешанных выражениях, включающих в себя как стандартные, так и пользовательские типы

данных. В этом случае нам самим придется создавать средства, выполняющие необходимые преобразования типов, поскольку компилятор не сможет преобразовывать пользовательские типы данных так, как он делал это со стандартными типами.

Явные преобразования типов

Явные преобразования типов, в отличие от неявных, совершаются самим программистом. Явные преобразования необходимы в тех случаях, когда компилятор не может безошибочно преобразовать типы автоматически.

В C++ существует несколько различных операций приведения типов, однако здесь мы ограничимся рассмотрением лишь одной из них.

Явные приведения типов в C++ ведут себя весьма требовательно. Вот пример оператора, осуществляющего преобразование типа int к типу char:

```
aCharVar = static_cast<char>(anIntVar);
```

Здесь переменная, тип которой мы хотим изменить, заключена в круглые скобки, а тип, к которому мы приводим переменную, — в угловые скобки. Приведение типа переменной anIntVar происходит перед присваиванием значения переменной aCharVar.

Вспомните, как в программе SIGNTEST мы получили неверный результат из-за слишком большого промежуточного значения переменной. Мы решили эту проблему путем использования типа unsigned int вместо int, поскольку в этом случае диапазон представления оказался достаточным для хранения вычисленного значения. А если бы и этого оказалось недостаточно? Тогда для решения проблемы можно использовать операцию приведения типов. Рассмотрим следующий пример.

```
// cast.cpp
// работа со знаковыми и беззнаковыми переменными
#include <iostream>
using namespace std;
int main()
{
  int intVar = 1500000000;        // 1 500 000 000
  intVar = (intVar * 10) / 10;    // слишком большой результат
  cout << "Значение intVar равно " << intVar << endl;
                                  // неверный результат
  intVar = 1500000000;            // приведение к типу double
  intVar = (static_cast<double>(intVar) * 10) / 10;
  cout << "Значение intVar равно " << intVar << endl;
                                   // верный результат
  return 0;
}
```

Когда мы умножаем переменную intVar на 10, получаемый результат, равный 15 000 000 000, нельзя хранить даже с помощью типа unsigned int, поскольку это приведет к получению неверного результата, подобного уже рассмотренному.

Конечно, мы могли бы изменить тип данных на double, которого было бы достаточно для хранения нашего числа, поскольку тип double позволяет хранить

числа длиной до 15 знаков. Но если мы не можем позволить себе подобный выход, например из-за небольшого количества имеющейся в наличии памяти, то существует другой способ — привести переменную intVar перед умножением к типу double. Оператор

```
static_cast<double>(intVar)
```

создает временную переменную типа double, содержащую значение, равное значению intVar. Затем эта временная переменная умножается на 10, и поскольку результат также имеет тип double, переполнения не происходит. Затем временная переменная делится на 10, и результат присваивается обычной целой переменной intVar. Результат работы программы выглядит следующим образом:

```
Значение intVar равно 211509811
Значение intVar равно 1500000000
```

Первый из результатов, полученный без приведения типов, неверен; второй результат, являющийся результатом работы с приведением типов, оказывается правильным.

До появления стандартного C++ приведение типов осуществлялось в несколько ином формате. Если сейчас оператор с приведением типов выглядит так:

```
aCharVar = static_cast<char>(anIntVar);
```

то раньше он записывался подобным образом:

```
aCharVar = (char)anIntVar;
```

или

```
aCharVar = char(anIntVar).
```

Недостаток последних двух форматов заключается в том, что их трудно найти в листинге как визуально, так и с помощью команды поиска редактора кода. Формат, использующий static_cast, проще обнаружить как одним, так и другим способом. Несмотря на то, что старые способы приведения типа до сих пор поддерживаются компиляторами, их употребление не рекомендуется.

Приведение типов следует использовать только в случае полной уверенности в его необходимости и понимания, для чего оно делается. Возможность приведения делает типы данных незащищенными от потенциальных ошибок, поскольку компилятор не может проконтролировать корректность действий при изменении типов данных. Но в некоторых случаях приведение типов оказывается совершенно необходимым, и мы убедимся в этом в наших будущих примерах.

Арифметические операции

Как вы уже, вероятно, заметили, в языке C++ используются четыре основные арифметические операции: сложения, вычитания, умножения и деления, обозначаемые соответственно +, -, *, /. Эти операции применимы как к целым типам данных, так и к вещественным, и их использование практически ничем не отличается ни

от других языков программирования, ни от алгебры. Однако существуют также и другие арифметические операции, использование которых не столь очевидно.

Остаток от деления

Существует еще одна арифметическая операция, которая применяется только к целым числам типа char, short, int и long. Эта операция называется *операцией остатка от деления* и обозначается знаком процента %. Результатом этой операции, иногда также называемой «взятием по модулю», является остаток, получаемый при делении ее левого операнда на правый. Программа REMAIND демонстрирует применение этой операции.

```
// remaind.cpp
// применение операции остатка от деления
#include <iostream>
using namespace std;
int main()
{
  cout <<6 %8 <<endl //6
  <<7 %8 <<endl      //7
  <<8 %8 <<endl      //0
  <<9 %8 <<endl      //1
  <<10 %8 <<endl;    //2
  return 0;
}
```

В этой программе берутся остатки от деления чисел 6, 7, 8 , 9 и 10 на 8, а результаты — 6, 7, 0, 1 и 2 — выводятся на экран. Операция остатка от деления используется довольно широко, и она еще понадобится нам в других программах.

Заметим, что в операторе

```
cout << 6 % 8
```

операция остатка от деления выполняется раньше, чем операция <<, поскольку приоритет последней ниже. Если бы это было не так, то мы были бы вынуждены заключить операцию 6 % 8 в скобки, чтобы выполнить ее до посылки в поток вывода.

Арифметические операции с присваиванием

Язык C++ располагает средствами для того, чтобы сократить размер кода и сделать его наглядным. Одним из таких средств являются *арифметические операции с присваиванием*. Они помогают придать характерный вид программному коду в стиле C++.

В большинстве языков программирования типичным является оператор, подобный

```
total = total + item; // сложение total и item
```

В данном случае вы производите сложение с замещением уже существующего значения одного из слагаемых. Но такая форма оператора не отличается кратко-

стью, поскольку нам приходится дважды использовать в нем имя **total**. В C++ существует способ сократить подобные операторы, применяя арифметические операции с присваиванием. Такие операции комбинируют арифметическую операцию и операцию присваивания, тем самым исключая необходимость использования имени переменной дважды. Предыдущий оператор можно записать с помощью сложения с присваиванием следующим образом:

```
total += item; // сложение total и item
```

На рис. 2.10 продемонстрирована эквивалентность указанных двух операторов.

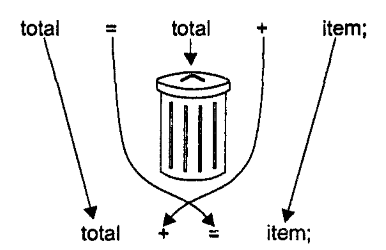


Рис. 2.10. Операция арифметического присваивания

С присваиванием комбинируется не только операция сложения, но и другие арифметические операции: -=, *=, /=, %= и т. д. Следующий пример демонстрирует использование арифметических операций с присваиванием:

```
// assign.cpp
// применение арифметических операций с присваиванием
#include <iostream>
using namespace std;
int main()
{
  int ans = 27;
  ans += 10;          // то же самое, что ans = ans +10;
  cout << ans << ", ";
  ans -= 7;           // то же самое, что ans = ans -7;
  cout << ans << ", ";
  ans *=2 ;           // то же самое, что ans = ans *2;
  cout << ans << ", ";
  ans /= 3;           // то же самое, что ans = ans /3;
  cout << ans << ", ";
  ans %= 3;           // то же самое, что ans = ans %3;
  cout << ans << endl;
  return 0;
}
```

Результат работы такой программы будет следующим:

```
37, 30, 60, 20, 2
```

Использовать арифметические операции с присваиванием при программировании не обязательно, но они являются довольно употребительными в C++, и мы будем пользоваться ими в других наших примерах.

Инкремент

Операция, которую мы сейчас рассмотрим, является более специфичной, нежели предыдущие. При программировании вам часто приходится иметь дело с увеличением какой-либо величины на единицу. Это можно сделать «в лоб», используя оператор

```
count = count + 1;   //увеличение count на 1
```

или с помощью сложения с присваиванием:

```
count += 1;          //увеличение count на 1
```

Но есть еще один, более сжатый, чем предыдущие, способ:

```
++count;             //увеличение count на 1
```

Операция ++ инкрементирует, или увеличивает на 1, свой операнд.

Префиксная и постфиксная формы

Знак операции инкремента может быть записан двояко: в префиксной форме, когда он расположен перед своим операндом, и в постфиксной форме, когда операнд записан перед знаком ++. В чем разница? Часто инкрементирование переменной производится совместно с другими операциями над ней:

```
totalWeight = avgWeight * ++count;
```

Возникает вопрос — что выполняется раньше: инкрементирование или умножение? В данном случае первым выполняется инкрементирование. Каким образом это определить? Префиксная форма записи и означает то, что инкремент будет выполнен первым. Если бы использовалась постфиксная форма, то сначала бы выполнилось умножение, а затем переменная count была бы увеличена на 1. Рисунок 2.11 иллюстрирует две указанные формы записи.

Рассмотрим пример, в котором используются как префиксная, так и постфиксная формы инкрементирования.

```
// increm.cpp
// применение операции инкрементирования
#include <iostream>
using namespace std;
int main()
{
  int count =10;
  cout << "count=" << count << endl;     // вывод числа 10
  cout << "count=" << ++count << endl;   // вывод числа 11 (префиксная форма)
  cout << "count=" << count << endl;     // вывод числа 11
  cout << "count=" << count++ << endl;   // вывод числа 11 (постфиксная форма)
  cout << "count=" << count << endl;     // вывод числа 12
  return 0;
}
```

При первом инкрементировании переменной count мы использовали префиксную форму для того, чтобы вывести на экран значение переменной, уже увеличенное на единицу. Поэтому второй оператор выводит на экран значение 11, а не 10.

Постфиксное инкрементирование, наоборот, действует после вывода на экран значения 11, и только следующий оператор получит значение count, равное 12.

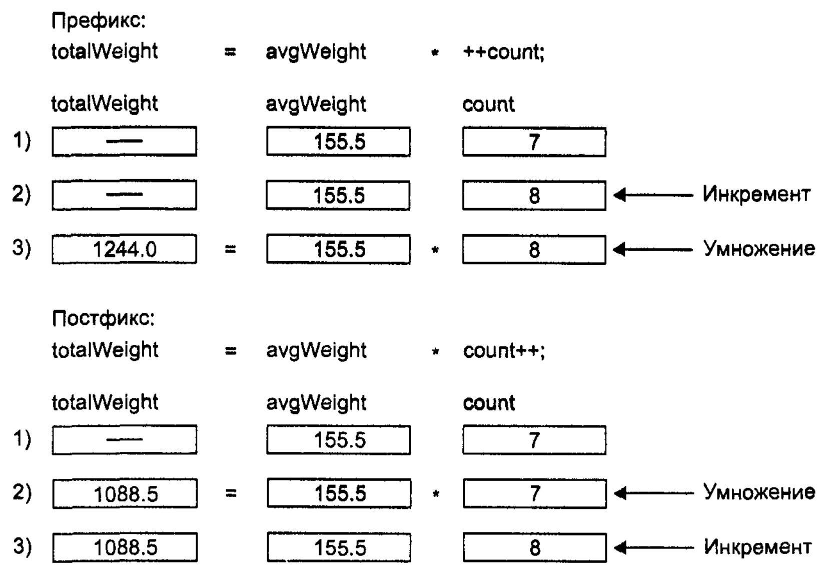


Рис. 2.11. Операция инкремента

Декремент

Операция декремента, обозначаемая --, в отличие от операции инкремента, уменьшает, а не увеличивает, на единицу свой операнд. Декремент также допускает префиксную и постфиксную формы записи.

Библиотечные функции

Многие действия языка C++ производятся с помощью библиотечных функций. Эти функции обеспечивают доступ к файлам, производят математические расчеты, выполняют преобразование данных и множество других действий. До тех пор пока мы не рассмотрим механизм работы функций в главе 5, нет особого смысла вдаваться в подробности относительно библиотечных функций. Тем не менее, уже сейчас вы сможете использовать наиболее простые из библиотечных функций на основе элементарных знаний об их работе.

В следующем примере под названием SQRT используется библиотечная функция sqrt(), вычисляющая значения квадратного корня из числа, вводимого пользователем.

```
// sqrt.cpp
// использование библиотечной функции sqrt()
#include <iostream>      //для cout и т.п.
```

```
#include <cmath>        //для sqrt()
using namespace std;
int main()
{
  double number,answer; //аргументы типа double для функции sqrt()
  cout << "Введите число: ";
  cin >> number;        // ввод числа
  answer = sqrt(number); // извлечение корня
  cout << "Квадратный корень равен "
  << answer << endl;    // вывод результата
  return 0;
}
```

Сначала программа получает значение от пользователя. Затем полученное значение используется в качестве аргумента для функции sqrt():

```
answer = sqrt(number);
```

Аргумент — это входные данные для функции. Аргумент заключается в круглые скобки, следующие после имени функции. Функция обрабатывает аргумент и возвращает значение — выходные данные функции. В данном случае возвращаемым значением функции является квадратный корень из ее аргумента. Возвращение значения функцией означает возможность присваивания такого значения переменной соответствующего типа, в данном случае — переменной answer. Затем программа отображает значение переменной на экране. Результат работы программы выглядит следующим образом:

```
Введите число: 1000
Квадратный корень равен 31.622777
```

Возведя в квадрат результат, выданный программой, с помощью обычного калькулятора, вы можете убедиться, что результат действительно верен.

Аргументы функций и возвращаемые ими значения должны иметь правильный тип. Информацию об этих типах можно получить с помощью справочной системы вашего компилятора, содержащей описания сотен библиотечных функций. Для функции sqrt() и аргумент, и возвращаемое значение должны иметь тип double, поэтому мы используем в программе переменные именно этого типа.

Заголовочные файлы

Подобно тому, как мы делали для cout и других объектов, необходимо включать в программу заголовочные файлы с описанием тех библиотечных функций, которые мы используем. В описании функции sqrt() мы можем прочитать, что ее описание содержится в файле CMATH. В программе SQRT содержится директива

```
#include <cmath>
```

которая и обеспечивает подключение файла CMATH. Если вы забудете подключить заголовочный файл с той библиотечной функцией, которую вы используете в программе, компилятор выдаст сообщение об ошибке.

Библиотечные файлы

Как мы уже говорили, при создании исполняемого файла к вашей программе будут прикомпонованы различные файлы, содержащие различные объекты и библиотечные функции. Такие файлы содержат исполняемый машинный код функций и часто имеют расширение .LIB. В одном из подобных файлов содержится и функция sqrt(). Она в нужный момент автоматически извлекается компоновщиком, который устанавливает необходимые связи так, что функцию можно вызывать из нашей программы. Компилятор берет на себя все необходимые действия, поэтому вам не нужно участвовать в процессе самому. Тем не менее, нелишним является понимание того, для чего нужны указанные файлы.

Заголовочные и библиотечные файлы

Давайте рассмотрим различия между заголовочными и библиотечными файлами. Для того чтобы использовать библиотечные функции, подобные sqrt(), необходимо связать библиотечный файл, содержащий эти функции, с вашей программой. Действия по подключению соответствующих функций к вашей программе осуществляются с помощью компоновщика.

Однако это еще не все. Функции вашей программы должны знать имена и типы функций и других элементов библиотечного файла. Эта информация и содержится в заголовочном файле. Каждый заголовочный файл содержит информацию об определенном наборе функций. Сами функции сгруппированы в библиотечных файлах, а информация о них содержится в заголовочных файлах. Так, например, файл IOSTREAM содержит описания объектов и функций, использующихся для ввода/вывода информации, в частности, объекта cout; файл CMATH содержит описания различных математических функций, таких, как sqrt(). Если бы вам понадобились функции для работы со строками, например strcpy(), тогда вам пришлось бы включить в свою программу заголовочный файл STRING.H и т. д.

На рис. 2.12 представлены отношения между заголовочными, библиотечными и прочими файлами, участвующими в создании программы.

Использование заголовочных файлов не является чем-то необычным в C++. В случаях, когда нужно использовать библиотечные функции, необходимо подключить заголовочные файлы, включающие в себя соответствующие описания.

Формы директивы #include

Существует две формы директивы #include. Угловые скобки < и >, в которые мы заключили имена файлов IOSTREAM и CMATH, указывают на то, что компилятор будет сначала искать эти файлы в стандартной директории с именем INCLUDE, как правило, содержащей заголовочные файлы для компилятора.

Вместо угловых скобок можно использовать и обычные двойные кавычки:

```
#include "myheader.h"
```

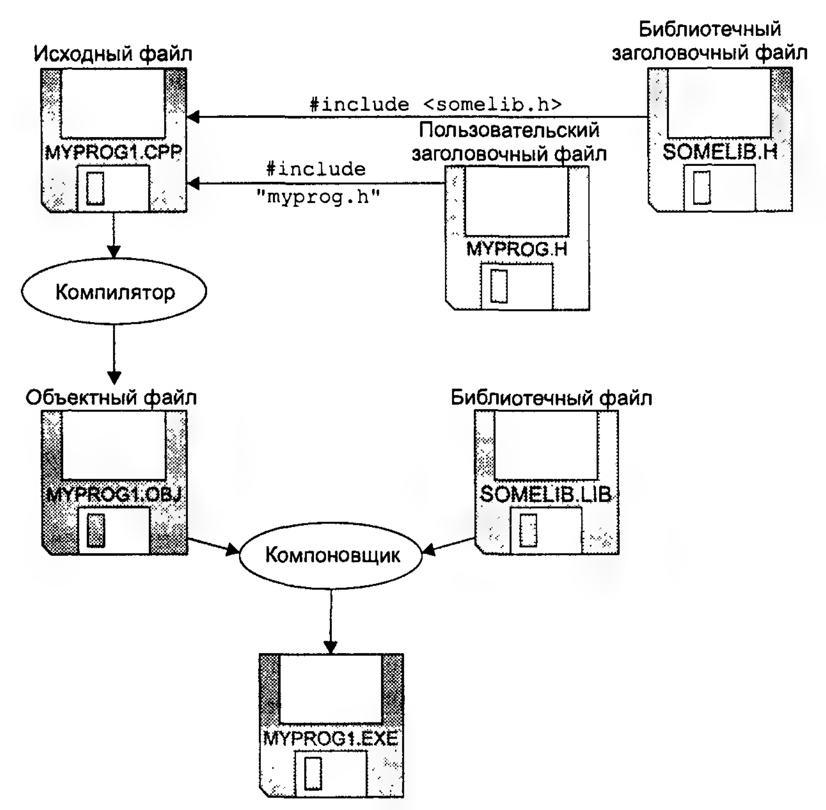


Рис. 2.12. Заголовочные и библиотечные файлы

Двойные кавычки указывают компилятору на то, что поиск файла нужно начинать с текущей директории. Обычно текущей директорией является та, в которой находится исходный файл. Двойные кавычки, как правило, используются для тех заголовочных файлов, которые вы создаете сами (подобная ситуация обсуждается в главе 13 «Многофайловые программы»).

Оба указанных способа являются вполне корректными для любого заголовочного файла, однако использование более подходящего из способов ускоряет процесс подключения, поскольку компилятор быстрее найдет нужный файл.

В приложениях В «Microsoft Visual C++» и Г «Borland C++ Builder» вы можете найти информацию о работе с заголовочными файлами при помощи одноименных компиляторов.

Резюме

В этой главе мы узнали, что главной составной частью программы на C++ является функция. Когда программа запускается на выполнение, функция с именем main() всегда исполняется первой.

Функция состоит из операторов, которые являются указаниями компьютеру выполнить соответствующие действия. Все операторы завершаются точкой с запятой (;). Оператор может содержать одно или более выражений, каждое из которых представляет собой совокупность переменных и операций над ними, приводящую к вычислению некоторого значения.

Вывод в C++, как правило, осуществляется с помощью объекта cout и операции вставки <<, при совместном использовании посылающих значения переменных и констант в стандартное устройство вывода, которым обычно является экран. Для ввода данных используется объект cin и операция извлечения >>, которые извлекают значения из стандартного устройства ввода — как правило, клавиатуры.

В C++ используются различные стандартные типы данных. char, int, long и short относятся к целым типам данных, а float, double и long double — к вещественным типам. Все указанные типы предусматривают наличие знака у числа. Беззнаковые типы данных, получаемые из стандартных с помощью ключевого слова unsigned, не предназначены для хранения отрицательных чисел, но способны хранить вдвое большие положительные числа. Для хранения булевых переменных используется тип bool, который имеет только два значения — true и false.

Ключевое слово const перед именем переменной запрещает изменять ее значение при выполнении программы. Строго говоря, такая переменная является константой.

В выражениях смешанного типа (то есть выражениях, содержащих данные различных типов) происходят автоматические приведения типов. Программист может также задать приведение типов явно.

C++ поддерживает четыре основных арифметических операции: +, -, * и /. Кроме того, существует операция остатка от деления %, возвращающая остаток от деления операнда на целое число.

Арифметические операции с присваиванием +=, -= и т. д. позволяют выполнять арифметическую операцию и присваивание ее результата одновременно. Операции инкремента ++ и декремента -- эквивалентны соответственно увеличению и уменьшению значения переменной на единицу.

Директивы препроцессора, в отличие от операторов, являются указаниями компилятору, а не компьютеру. Директива #include указывает компилятору включить текст указываемого файла в исходный файл, а директива #define — заменить в тексте программы первое указываемое в ней значение на второе. Директива using указывает, что все имена в тексте программы относятся к одному пространству имен.

Коды библиотечных функций, используемых в программе, содержатся в библиотечном файле и автоматически прикомпоновываются к программе. Заголовочный файл, содержащий описания соответствующих библиотечных функций, обязательно должен быть включен в текст исходной программы с помощью директивы #include.

Вопросы

Ответы на приведенные ниже контрольные вопросы можно найти в приложении Ж.

1. Разделение программы на функции:
   а) является ключевым методом объектно-ориентированного программирования;
   б) упрощает представление программы;
   в) сокращает размер программного кода;
   г) ускоряет процесс выполнения программы.
2. После имени функции ставятся ________.
3. Тело функции заключается в _________.
4. *В чем особенность функции main()?*
5. Конструкция C++, указывающая компьютеру выполнить действие, называется __________.
6. Напишите пример комментария в стиле C++ и пример устаревшего комментария /*.
7. Выражение:
   а) всегда приводит к вычислению значения;
   б) является способом высказывания программы;
   в) всегда происходит вне функции;
   г) является частью оператора.
8. Укажите размер переменных следующих типов в 32-битной системе:
   а) тип int;
   б) тип long double;
   в) тип float;
   г) тип long.
9. Истинно ли следующее утверждение: переменная типа char может хранить значение 301?
10. Укажите, к каким элементам программы относятся следующие:
    а) 12;
    б) 'a';
    в) 4.28915;
    г) JungleJim;
    д) JungleJim().
11. Напишите операторы, выводящие на экран:
    а) значение переменной x;

б) имя Jim;

в) число 509.

12. Истинно ли следующее утверждение: в операции присваивания величина, стоящая слева от знака равенства, всегда равна величине, стоящей справа от знака равенства?
13. Напишите оператор, выводящий значение переменной george в поле размером 10 символов.
14. Какой заголовочный файл нужно включить в исходный текст, чтобы использовать объекты cin и cout?
15. Напишите оператор, который получает с клавиатуры числовое значение и присваивает его переменной temp.
16. Какой заголовочный файл нужно включить в исходный текст, чтобы использовать манипулятор setw?
17. Двумя случаями, когда компилятор обрабатывает разделительные символы, являются ____________ и ____________.
18. Верно или неверно следующее утверждение: нет никаких препятствий к использованию переменных разного типа в одном арифметическом выражении?
19. Значение выражения 11 % 3 равно ____.
20. Действия каких двух типов операций сочетают в себе операции арифметического присваивания?
21. Напишите оператор, увеличивающий значение переменной temp на 23 с одновременным присваиванием. Напишите аналогичный оператор, не использующий сложения с присваиванием.
22. На какую величину увеличивает значение переменной операция инкремента?
23. Какие значения выведут на экран два указанных оператора, если начальное значение переменной var1 равно 20?

```
cout << var1--;
cout << ++var1;
```

24. С какой целью мы включали заголовочные файлы в тексты наших примеров?
25. Коды библиотечных функций содержатся в __________ файлах.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Считая, что кубический фут равен 7,481 галлона, написать программу, запрашивающую у пользователя число галлонов и выводящую на экран эквивалентный объем в кубических футах.

*2. Напишите программу, выводящую следующую таблицу:

```
1990      135
1991     7290
1992    11300
1993    16200
```

В программе использовать только один оператор с cout.

*3. Напишите программу, генерирующую следующий вывод:

```
10
20
19
```

Используйте представление числа 10 в виде целой константы. Для вывода числа 20 воспользуйтесь одной из арифметических операций с присваиванием, а для вывода числа 19 — операцией декремента.

4. Напишите программу, выводящую на экран ваше любимое стихотворение. Для разбиения на строчки используйте подходящую управляющую последовательность.

5. Библиотечная функция islower() принимает в качестве аргумента один символ (букву) и возвращает ненулевое целое значение в том случае, если буква является строчной, и нулевое, если буква является заглавной. Описание функции хранится в файле CTYPE.H. Напишите программу, которая принимает букву от пользователя, а затем выводит нулевое или ненулевое значение в зависимости от того, является ли буква строчной или нет.

6. На биржевых торгах за 1 фунт стерлингов давали $1.487, за франк — $0.172, за немецкую марку — $0.584, а за японскую йену — $0.00955. Напишите программу, которая запрашивает денежную сумму в долларах, а затем выводит эквивалентные суммы в других валютах.

7. Температуру, измеренную в градусах по Цельсию, можно перевести в градусы по Фаренгейту путем умножения на 9/5 и сложения с числом 32. Напишите программу, запрашивающую температуру в градусах по Цельсию и отображающую ее эквивалент по Фаренгейту.

8. Когда размер величины, выводимой на экран с помощью манипулятора setw(), оказывается меньше размера зарезервированного поля, по умолчанию незаполненные поля заполняются пробелами. Манипулятор setfill() принимает в качестве аргумента один символ, который замещает все пробелы на незаполненных позициях поля. Модифицируйте пример WIDTH так, чтобы символы, разделяющие пары значений из столбцов, были не пробелами, а точками, например

```
Москва.....8425785
```

9. Две дроби a/b и c/d можно сложить следующим образом:

a/b + c/d = (a*d + b*c)/(b*d)

Например, 1/4 + 2/3 = (1*3 + 4*2)/4*3 = 11/12

Напишите программу, запрашивающую у пользователя значения двух дробей, а затем выводящую результат, также записанный в форме дроби. Взаимодействие программы с пользователем может выглядеть, например, следующим образом:

```
Введите первую дробь: 1/2
Введите вторую дробь: 2/5
Сумма равна 9/10
```

Вы можете использовать тот факт, что операция извлечения **>>** может считывать более одного значения за раз:

```
cin >> a >> dummychar >> b;
```

10. Устаревшая денежная система Великобритании состояла из фунтов, шиллингов и пенсов. 1 фунт был равен 20 шиллингам, а 1 шиллинг — 12 пенсам. Для записи использовалась система, состоящая из знака £ и трех десятичных значений, разделенных точками. Например, запись £5.2.8 обозначала 5 фунтов, 2 шиллинга и 8 пенсов (пенс — множественное число от пенни). Современная денежная система, принятая в 50-е годы XX века, состоит только из фунтов и пенсов, причем один фунт равен 100 пенсам. Такой фунт называют десятичным. Таким образом, в новой денежной системе указанная сумма будет обозначаться как £5.13 (если быть точнее, £5.1333333). Напишите программу, которая будет преобразовывать сумму, записанную в старом формате (фунты, шиллинги, пенсы), в новый формат (фунты, пенсы). Форматом взаимодействия программы с пользователем может являться следующий:

```
Введите количество фунтов: 7
Введите количество шиллингов: 17
Введите количество пенсов: 9
Десятичных фунтов: £7.89
```

В большинстве компиляторов для представления знака £ используется десятичный код 156. Некоторые компиляторы позволяют скопировать знак фунта прямо из таблицы символов Windows.

11. По умолчанию форматирование вывода производится по правому краю поля. Можно изменить форматирование текста на левостороннее путем использования манипулятора setiosflags(ios::left) (не беспокойтесь о смысле новой формы записи, встретившейся в манипуляторе). Используйте этот манипулятор вместе с setw() для того, чтобы произвести следующий вывод:

```
Фамилия  Имя      Адрес          Город
-------------------------------------------------------
Петров   Василий  Кленовая 16    Санкт-Петербург
Иванов   Сергей   Осиновая 3     Находка
Сидоров  Иван     Березовая 21   Калининград
```

12. Напишите программу, выполняющую действия, обратные тем, которые описаны в упражнении 10, то есть запрашивающую у пользователя сумму, указанную в десятичных фунтах, и переводящую ее в старую систему

фунтов, шиллингов и пенсов. Пример взаимодействия программы с пользователем может выглядеть так:

```
Введите число десятичных фунтов: 3.51
Эквивалентная сумма в старой форме записи: £3.10.2
```

Обратите внимание на то, что если вам придется присваивать вещественное значение (например, 12,34) переменной целого типа, то его дробная часть (0,34) будет потеряна, а целая переменная получит значение 12. Чтобы избежать предупреждения со стороны компилятора, используйте явное преобразование типов. Можно использовать операторы, подобные приведенным ниже:

```
float decpounds;        // сумма в десятичных фунтах
int pounds;             // сумма в старых фунтах
float decfrac;          // десятичная дробная часть
pounds = static_cast<int>(decpounds); //отбрасывание
                                      //дробной части
decfrac = decpounds - pounds;         //прибавление дробной части
```

Чтобы получить число шиллингов, следует умножить на 20 значение переменной decfrac. Аналогичным образом можно получить число пенсов.

Глава 3
Циклы и ветвления

- Операции отношения
- Циклы
- Ветвления
- Логические операции
- Приоритеты операций C++
- Другие операторы перехода

Лишь немногие из программ выполняются последовательно от первого оператора к последнему. Как и большинство людей, программы определяют порядок своих действий в зависимости от меняющихся обстоятельств. В программе предусмотрены переходы из одной части программы в другую в зависимости от выполнения или невыполнения некоторого условия. Операторы, реализующие подобные переходы, называются *условными операторами*. Условные операторы делятся на две основные категории: *циклы* и *ветвления*.

Количество выполнений цикла, а также выполнение или невыполнение определенной части программы зависит от истинности или ложности вычисляемого выражения. Как правило, такие выражения содержат особый тип операций, называемых *операциями отношения*, сравнивающими два значения. Поскольку и циклы, и ветвления тесно связаны с операциями отношения, то в первую очередь мы займемся рассмотрением последних.

Операции отношения

Операция отношения сравнивает между собой два значения. Значения могут быть как стандартных типов языка C++, например char, int или float, так и типов, определяемых пользователем, как мы позже убедимся. Сравнение устанавливает одно из трех возможных отношений между переменными: *равенство*, *больше* или *меньше*. Результатом сравнения является значение *истина* или *ложь*. Например, две величины могут быть равны (истина) или не равны (ложь).

Наш первый пример RELAT демонстрирует использование операций сравнения применительно к целым переменным и константам.

```
// relat.cpp
// применение операций отношения
#include <iostream>
using namespace std;
int main()
{
  int numb;
  cout << "Введите число: ";
  cin >> numb;
  cout << "numb<10 равно " << (numb <10) << endl;
  cout << "numb>10 равно " << (numb >10) << endl;
  cout << "numb==10 равно " << (numb ==10) << endl;
  return 0;
}
```

Эта программа использует три вида сравнения числа 10 с числом, вводимым пользователем. Если пользователь введет значение, равное 20, то результат работы программы будет выглядеть так:

```
Введите число: 20
numb<10 равно 0
numb>10 равно 1
numb==10 равно 0
```

Первое выражение истинно в том случае, если значение numb меньше, чем 10; второе — тогда, когда numb больше, чем 10; и наконец, третье — когда numb равно 10. Как можно видеть из результата работы программы, компилятор C++ присваивает истинному выражению значение 1, а ложному — значение 0.

Как мы упоминали в предыдущей главе, стандартный C++ включает в себя тип bool, переменные которого имеют всего два значения: true и false. Возможно, вы подумали о том, что результаты сравнения выражений должны иметь тип bool, и, следовательно, программа должна выводить false вместо 0 и true вместо 1, но C++ интерпретирует true и false именно как 1 и 0. Это отчасти объясняется историческими причинами, поскольку в языке C не было типа bool, и наиболее приемлемым способом представить ложь и истину представлялись именно числа 0 и 1. С введением типа bool два способа интерпретации истинности и ложности объединились, и их можно представлять как значениями true и false, так и значениями 1 и 0. В большинстве случаев не важно, каким из способов пользоваться, поскольку нам чаще приходится использовать сравнения для организации циклов и ветвлений, чем выводить их результаты на экран.

Полный список операций отношения в C++ приводится ниже.

Операция	Название
>	больше
<	меньше
==	равно
!=	не равно
>=	больше или равно
<=	меньше или равно

Теперь рассмотрим выражения, использующие операции сравнения, и значения, получающиеся в результате вычисления таких выражений. В первых двух строках примера, приведенного ниже, присваиваются значения переменным harry и jane. Далее вы можете, закрыв комментарии справа, проверить свое умение определять истинность выражений.

```
jane = 44;           // оператор присваивания
harry = 12;          // оператор присваивания
(jane == harry)      // ложь
(harry <= 12)        // истина
(jane > harry)       // истина
(jane >= 44)         // истина
(harry != 12)        // ложь
(7 < harry)          // истина
(0)                  // ложь (по определению)
(44)                 // истина (поскольку не ноль)
```

Обратите внимание на то, что операция равенства, в отличие от операции присваивания, обозначается с помощью двойного знака равенства. Распространенной ошибкой является употребление одного знака равенства вместо двух. Подобную ошибку трудно распознать, поскольку компилятор не увидит ничего неправильного в использовании операции присваивания. Разумеется, эффект от работы такой программы, скорее всего, будет отличаться от желаемого.

Несмотря на то, что C++ использует 1 для представления истинного значения, любое отличное от нуля число будет воспринято им как истинное. Поэтому истинным будет считаться и последнее выражение в нашем примере.

Давайте теперь рассмотрим, как можно использовать указанные операции в различных типовых ситуациях. Сначала мы займемся организацией циклов, а затем перейдем к ветвлениям.

Циклы

Действие циклов заключается в последовательном повторении определенной части вашей программы некоторое количество раз. Повторение продолжается до тех пор, пока выполняется соответствующее условие. Когда значение выражения, задающего условие, становится ложным, выполнение цикла прекращается, а управление передается оператору, следующему непосредственно за циклом.

В C++ существует 3 типа циклов: for, while и do.

Цикл for

Большинство изучающих язык C++ считают цикл for самым легким для понимания. Все элементы, контролирующие его выполнение, собраны в одном месте, в то время как в циклах других типов они разбросаны внутри цикла, что зачастую делает логику его работы трудной для понимания.

Цикл for организует выполнение фрагмента программы фиксированное число раз. Как правило (хотя и не всегда), этот тип цикла используется тогда, когда число раз, за которое должно повториться исполнение кода, известно заранее.

В примере FORDEMO, приведенном ниже, выводятся на экран квадраты целых чисел от 0 до 14:

```
// fordemo.cpp
// демонстрирует работу простейшего цикла for
#include <iostream>
using namespace std;
int main()
{
  int j;                        // определение счетчика цикла
  for ( j=0 ; j<15 ; j++)       // счетчик меняется от 0 до 14
  cout << j * j << "  ";        // квадрат значения j выводится на экран
  cout << endl;
  return 0;
}
```

Результат работы программы выглядит так:

```
0  1  4  9  16  25  36  49  64  81  100  121  144  169  196
```

Каким образом работает эта программа? Оператор for управляет циклом. Он состоит из ключевого слова for, за которым следуют круглые скобки, содержащие три выражения, разделенные точками с запятой:

```
for (j=0 ; j<15 ; j++)
```

Первое из трех выражений называют *инициализирующим*, второе — *условием проверки*, а третье — *инкрементирующим*, как показано на рис. 3.1.

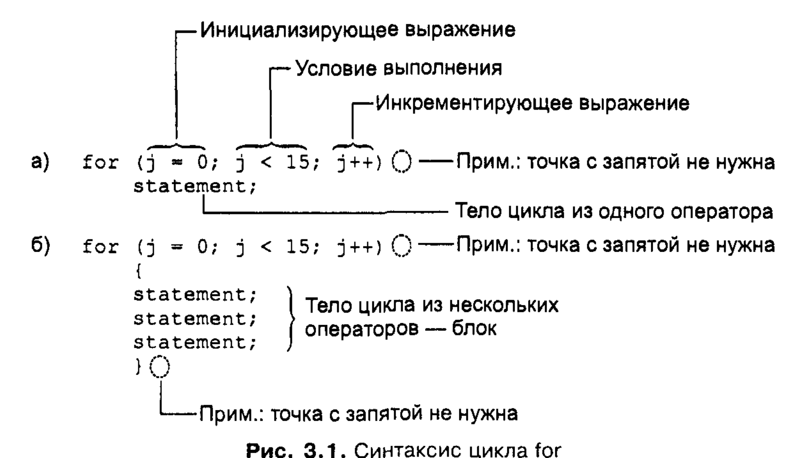


Рис. 3.1. Синтаксис цикла for

Эти три выражения, как правило (но не всегда), содержат одну переменную, которую обычно называют *счетчиком цикла*. В примере FORDEMO счетчиком цикла является переменная j. Она определяется до того, как начнет исполняться тело цикла.

Под телом цикла понимается та часть кода, которая периодически исполняется в цикле. В нашем примере тело цикла состоит из единственного оператора

```
cout << j * j << "  ";
```

Данный оператор печатает квадрат значения переменной j и два пробела после него. Квадрат находится как произведения переменной j самой на себя. Во время исполнения цикла переменная j принимает значения 0, 1, 2, 3 и т. д. до 14, а выводимые значения квадратов соответственно 0, 1, 4, 9, ..., 196.

Обратите внимание на то, что после оператора for отсутствует точка с запятой (;). Это объясняется тем, что на самом деле оператор for вместе с телом цикла представляют из себя один оператор. Это очень важная деталь, поскольку если поставить после оператора for точку с запятой, то компилятор воспримет это как отсутствие тела цикла, и результат работы программы будет отличаться от задуманного.

Рассмотрим, каким образом три выражения, стоящие в круглых скобках оператора for, влияют на работу цикла.

Инициализирующее выражение

Инициализирующее выражение вычисляется только один раз — в начале выполнения цикла. Вычисленное значение инициализирует счетчик цикла. В примере FORDEMO переменная j получает значение 0.

Условие выполнения цикла

Как правило, условие выполнения цикла содержит в себе операцию отношения. Условие проверяется каждый раз перед исполнением тела цикла и определяет, нужно ли исполнять цикл еще раз или нет. Если условие выполняется, то есть соответствующее выражение истинно, то цикл исполняется еще раз. В противном случае управление передается тому оператору, который следует за циклом. В программе FORDEMO после завершения цикла управление передается оператору

```
cout << endl;
```

Инкрементирующее выражение

Инкрементирующее выражение предназначено для изменения значения счетчика цикла. Часто такое изменение сводится к инкрементированию счетчика. Модификация счетчика происходит после того, как тело цикла полностью выполнилось. В нашем примере увеличение j на единицу происходит каждый раз после завершения тела цикла.

Блок-схема, иллюстрирующая исполнение цикла for, приведена на рис. 3.2.

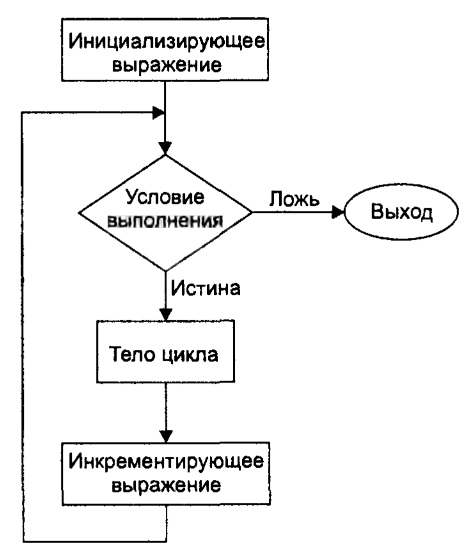


Рис. 3.2. Исполнение цикла for

Число выполнений цикла

В примере FORDEMO цикл выполняется 15 раз. В первый раз он выполняется при нулевом значении j. Последнее исполнение цикла произойдет при j=14, поскольку условием выполнения цикла служит j<15. Когда j принимает значение, равное 15, выполнение цикла прекращается. Как правило, подобную схему исполнения применяют в том случае, когда необходимые действия нужно выполнить фиксированное количество раз. Присваивая счетчику начальное значение, равное 0, в качестве условия продолжения цикла ставят сравнение счетчика с желаемым числом выполнений и увеличивают счетчик на единицу каждый раз, когда исполнение тела цикла завершается. Например, цикл

```
for (count = 0; count < 100; count++)
//тело цикла
```

будет выполнен 100 раз, а счетчик count будет принимать значения от 0 до 99.

Несколько операторов в теле цикла

Разумеется, вам может понадобиться выполнить в теле цикла не один, а несколько операторов. Тогда эти несколько операторов необходимо заключить в фигурные скобки, как мы поступали с телом функций. Обратите внимание на то, что после закрывающей фигурной скобки не следует ставить точку с запятой подобно тому, как мы делаем в конце операторов, входящих в тело цикла.

Следующий пример с названием CUBELIST демонстрирует использование нескольких операторов в теле одного цикла. Программа возводит в куб числа от 1 до 10 и печатает результаты в двух столбцах.

```
// cubelist.cpp
// подсчет кубов целых чисел от 1 до 10
#include <iostream>
#include <iomanip>                    //для setw
using namespace std;
int main()
{
  int numb;                           // счетчик цикла
  for(numb=1; numb<=10; numb++)       // цикл от 1 до 10
  {
    cout << setw(4) << numb;          // вывод первого столбца
    int cube = numb * numb * numb;    // вычисление куба
    cout << setw(6) << cube << endl;  // вывод второго столбца
  }
  return 0;
}
```

Результат работы программы выглядит следующим образом:

```
1   1
2   8
3   27
4   64
5   125
6   216
7   343
8   512
9   729
10  1000
```

Мы показали, что оператор цикла совсем не обязательно использовать так, как мы делали в предыдущем примере: начальное значение у счетчика здесь 1, а не 0; увеличение счетчика происходит до 10, а не до 9, а условие проверки содержит операцию сравнения «больше или равно» <=. Таким образом, цикл исполняется 10 раз, а счетчик пробегает значения от 1 до 10 (а не от 0 до 9).

Обратите внимание на то, что даже если в теле вашего цикла содержится всего один оператор, вы также можете заключить его в фигурные скобки, хотя делать это не обязательно. Некоторые программисты предпочитают всегда заключать тело цикла в фигурные скобки, мотивируя это соображениями удобства восприятия программного кода.

Блоки и область видимости переменных

Тело цикла, заключенное в фигурные скобки, называется *блоком*. Важной особенностью блока является то, что переменные, определенные внутри него, *невидимы* вне этого блока. Невидимость означает, что программа не имеет доступа к переменной (мы подробнее остановимся на понятии видимости в главе 5 «Функции»). В примере CUBELIST мы определяем переменную cube внутри блока:

```
int cube = numb * numb * numb;
```

Получить доступ к этой переменной вне блока невозможно — она видима лишь внутри фигурных скобок. Если вы попытаетесь присвоить переменной cube значение вне блока

```
cube = 10;
```

то компилятор выдаст сообщение о том, что переменная с именем cube *не определена.*

Преимуществом такого ограничения области видимости переменных является то, что одно и то же имя переменной можно использовать несколько раз в разных блоках программы (определение переменной внутри блока распространено в C++, но редко используется в C).

Форматирование и стиль оформления циклов

Хороший стиль программирования предполагает сдвиг тела цикла вправо относительно оператора, управляющего циклом, и относительно остального программного кода. В программе FORDEMO такой сдвиг применен к одной строке, а в программе CUBELIST вправо сдвинут весь блок, за исключением обрамляющих фигурных скобок. Подобное форматирование является очень удобным, поскольку позволяет легко увидеть, где начинается цикл, а где заканчивается. Компилятор не отличает форматированный текст от неформатированного (по крайней мере, на его работе это никак не сказывается).

Существует еще одна разновидность того стиля, который мы применяли для оформления циклов в наших программах. Мы выравнивали фигурные скобки по вертикали, но некоторые программисты предпочитают помещать открывающую фигурную скобку прямо после оператора цикла, как показано здесь:

```
for(numb=1;numb<=10;numb++){
  cout <<setw(4)<<numb;
  int cube =numb*numb*numb;
  cout <<setw(6)<<cube <<endl;
  }
```

Такой способ делает листинг на одну строку короче, но восприятие текста становится менее удобным, поскольку открывающая фигурная скобка теряется из виду и путаются связи между открывающими и закрывающими фигурными скобками. Еще одним вариантом оформления циклов служит сдвиг вправо тела цикла вместе с обрамляющими фигурными скобками:

```
for(numb=1;numb<=10;numb++)
  {
  cout <<setw(4)<<numb;
  int cube =numb*numb*numb;
  cout <<setw(6)<<cube <<endl;
    }
```

Этот подход является весьма распространенным, однако и у него есть свои противники. Разумеется, выбор способа оформления кода программы зависит только от ваших предпочтений, и вы можете форматировать текст программы так, как вам кажется удобнее.

Обнаружение ошибок

С помощью средств компилятора, позволяющих облегчить процесс обнаружения ошибок в программах, вы можете создать динамическую модель, иллюстрирующую процесс выполнения вашего цикла. Главным из таких средств является *пошаговое выполнение*. Откройте окно проекта для отлаживаемой программы и окно с текстом программы. Детали работы с отладчиком зависят от компиляторов, информацию о которых можно получить из приложения В «Microsoft Visual C++» или приложения Г «Borland C++ Builder». Нажимая соответствующую функциональную клавишу, можно построчно исполнять код вашей программы. Таким образом, вы сможете увидеть работу программы в том порядке, в котором записаны ваши операторы. При работе с циклом вы сможете убедиться в том, что сначала исполняется тело вашего цикла, затем происходит переход, и тело цикла исполняется снова.

Отладчик можно также использовать для того, чтобы следить за значениями переменных в процессе исполнения программы. Вы можете поэкспериментировать с программой cubelist, поместив переменные numb и cube в окно Watch window вашего отладчика, и посмотреть на изменения их значений при исполнении программы. Чтобы получить информацию об использовании окна Watch, загляните в соответствующее приложение.

Watch window и пошаговое исполнение программы являются мощным отладочным инструментом. Если поведение вашей программы отличается от задуманного, вы можете использовать эти средства для контроля над объектами программы. Как правило, причина ошибки после такой отладки становится ясной.

Варианты цикла for

Инкрементирующий оператор не обязательно должен производить операцию инкрементирования счетчика цикла; вместо инкрементирования может использоваться любая другая операция. В следующем примере под названием FACTOR в операторе цикла используется декрементирование счетчика цикла. Программа запрашивает значение у пользователя, а затем подсчитывает факториал этого числа (факториал числа представляет из себя произведение всех целых положительных чисел, не превышающих данное число. Например, факториал числа 5 равен *1*2*3*4*5=120*).

```
// factor.cpp
// подсчет факториала числа с помощью цикла for
#include <iostream>
using namespace std;
int main()
{
  unsigned int numb;
  unsigned long fact = 1;        // тип long для результата
  cout << "Введите целое число: ";
  cin >> numb;                   // ввод числа
  for(int j=numb; j>0; j--)      // умножение 1 на
  fact *= j;                     // numb, numb-1, ..., 2, 1
```

```
  cout << "Факториал числа равен " << fact << endl;
  return 0;
}
```

В этом примере инициализирующий оператор присваивает переменной j значение, вводимое пользователем. Условием продолжения цикла является положительность значения j. Инкрементирующее выражение после каждой итерации уменьшает значение j на единицу.

Мы использовали тип unsigned long для переменной, хранящей значение факториала, потому, что даже для небольших чисел значения их факториалов очень велико. В 32-битных системах размеры типов int и long совпадают, но в 16-битных системах размер типа long больше. Следующий результат работы программы показывает, насколько велико может быть значение факториала даже для небольшого числа:

```
Введите целое число: 10
Факториал числа равен 3628800
```

Самое большое число, которое можно использовать для ввода в этой программе, равно 12. Для чисел больше 12 результат работы программы будет неверен, поскольку произойдет переполнение.

Определение счетчика цикла внутри оператора цикла for

В последней программе есть еще одно полезное нововведение: переменная j описана прямо внутри оператора цикла:

```
for (int j = numb; j>0; j--)
```

Подобная конструкция является типичной для C++, и, как правило, наиболее удобна для работы со счетчиками цикла. Такое определение переменной стоит наиболее близко к месту ее употребления. Переменная, описанная в операторе цикла, видна только внутри этого цикла (компилятор Microsoft делает такие переменные видимыми от точки объявления до конца программы, но это не является стандартом C++).

Несколько инициализирующих выражений и условий цикла

Вместо одного инициализирующего выражения в операторе цикла for можно использовать несколько выражений, разделяемых запятыми. Подобным же образом можно использовать более одного инкрементирующего выражения. Лишь условие продолжения цикла всегда должно быть одно. Приведем такой пример:

```
for (j = 0, alpha = 100; j <50; j++, beta--)
{
// тело цикла
}
```

У данного цикла есть обычный счетчик в виде переменной j, но в операторе цикла, помимо j, также инициализируется переменная alpha и декрементируется переменная beta. Переменные alpha и beta никак не связаны ни друг с другом, ни с переменной j. При использовании нескольких инициализирующих или инкрементирующих выражений необходимо разделять их запятыми.

Из трех выражений, используемых при задании цикла, на самом деле ни одно не является обязательным. Так, например, конструкция

```
for ( : : )
```

эквивалентна циклу while с условием продолжения, равным true. Мы рассмотрим циклы while чуть позже.

В дальнейшем мы не будем использовать ни множественные выражения в операторе цикла, ни их отсутствие. Несмотря на то, что подобные конструкции сокращают листинг программы, читаемость кода при этом страдает. Всегда можно организовать цикл так, что оператор цикла будет содержать в себе только три оператора, указанные вначале.

Цикл while

Цикл for выполняет последовательность действий определенное количество раз. А как поступить в том случае, если заранее не известно, сколько раз понадобится выполнить цикл? Для этого разработан другой вид цикла — while.

В следующем примере под названием ENDONO пользователю предлагают ввести серию значений. В том случае, когда вводимое значение оказывается равным нулю, происходит выход из цикла. Очевидно, что в этой ситуации заранее невозможно узнать, сколько ненулевых значений введет пользователь.

```
// endon0.cpp
// применение цикла WHILE
#include <iostream>
using namespace std;
int main()
{
  int n = 99;            // n не должна быть равна 0 перед началом цикла
  while(n != 0 )         // цикл, пока значение n не равно 0
  cin >> n;              // считывание n с клавиатуры
  cout << endl;
  return 0;
}
```

Далее мы приведем возможный пример работы программы. Пользователь вводит значения, а программа отображает эти значения до тех пор, пока пользователь не введет ноль, после чего программа завершается.

```
1
27
33
144
9
0
```

Внешне цикл while напоминает упрощенный вариант цикла for. Он содержит условие для продолжения цикла, но не содержит ни инициализирующих, ни инкрементирующих выражений. Синтаксис цикла while показан на рис. 3.3.

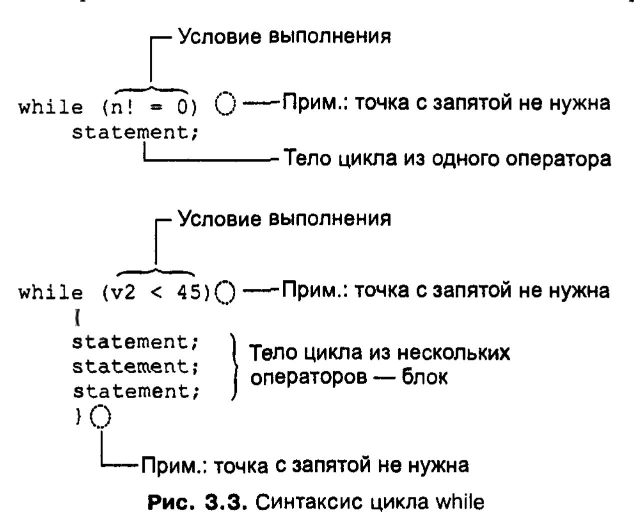


Рис. 3.3. Синтаксис цикла while

До тех пор пока условие продолжения цикла выполняется, исполнение тела цикла продолжается. В примере ENDONO значение выражения

```
n != 0
```

истинно до тех пор, пока пользователь не введет ноль.

На рис. 3.4 показан механизм работы цикла while. На самом деле он не так прост, как кажется вначале. Несмотря на отсутствие инициализирующего оператора, нужно инициализировать переменную цикла до начала исполнения тела цикла. Тело цикла должно содержать оператор, изменяющий значение переменной цикла, иначе цикл будет бесконечным. Таким оператором в цикле из примера ENDONO является cin >> n;

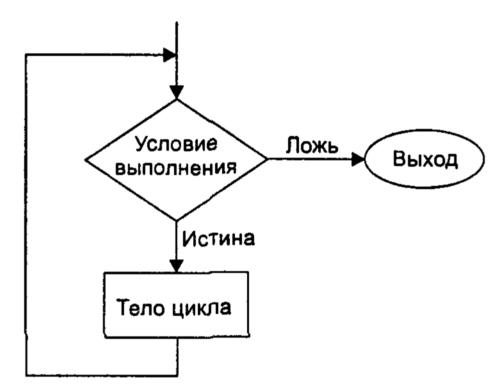


Рис. 3.4. Исполнение цикла while

Несколько операторов в цикле while

Следующий пример, WHILE4, демонстрирует применение нескольких операторов в теле цикла while. Это немного модифицированная версия программы CUBELIST, которая вычисляет не третьи, а четвертые степени последовательности целых чисел. Поставим дополнительное условие к задаче: все выводимые значения должны иметь размер не более 4 символов, то есть необходимо завершить выполнение цикла, когда результат превысит 9999. Без предварительного расчета мы не знаем, какое число будет источником такого результата, поэтому возложим определение этого числа на нашу программу. Условие продолжения цикла будет сформировано так, что программа завершится до того, как результаты превысят установленный барьер.

```
// while4.cpp
// возведение в четвертую степень целых чисел
#include <iostream>
#include <iomanip>                //для setw
using namespace std;
int main()
{
  int pow = 1;                    // первое возводимое число равно 1
  int numb = 1;                   // 1 в 4-й степени равна 1
  while(pow < 10000)              // цикл, пока в степени
                                  // не более 4 цифр
  {
    cout << setw(2) << numb;      // вывод числа
    cout << setw(5) << pow <<endl; // и его 4-й степени
    ++numb;                       // инкремент текущего числа
    pow = numb*numb*numb*numb;    // вычисление 4-й степени
  }
  cout << endl;
  return 0;
}
```

Для того чтобы найти значение четвертой степени числа, мы просто умножаем это число само на себя четырежды. После каждой итерации мы увеличиваем значение переменной numb на единицу, но нам не нужно использовать значение numb в операторе цикла while, поскольку выполнение цикла зависит только от значения результирующей переменной pow. Результат работы программы выглядит так:

```
1    1
2   16
3   81
4  256
5  625
6 1296
7 2401
8 4096
9 6561
```

Следующее значение — 10 000 — будет слишком велико для вывода в четыре позиции, но к этому времени уже будет произведен выход из программы.

Приоритеты арифметических операций и операций отношения

Следующая программа затронет вопрос старшинства операций. В этой программе генерируется последовательность чисел Фибоначчи. Первыми членами такой последовательности являются числа

```
1 1 2 3 5 8 13 21 34 55
```

Каждый новый член получается путем сложения двух предыдущих: 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8 и т. д. Числа Фибоначчи проявляют себя в таких различных ситуациях, как методы сортировки и подсчет числа спиралей у цветков подсолнуха.

Одним из наиболее интересных применений чисел Фибоначчи является их связь с так называемым «золотым сечением». Золотое сечение — это идеальные пропорции в архитектуре и искусстве, воплощенные при строительстве древнегреческих храмов. Чем больше номера членов последовательности чисел Фибоначчи, тем ближе отношение последних двух членов к золотому сечению. Текст программы FIBO приведен ниже.

```
// fibo.cpp
// генерирование чисел Фибоначчи с помощью цикла while
#include <iostream>
using namespace std;
int main()
{
//граница типа unsigned long
const unsigned long limit = 4294967295;
unsigned long next = 0;      // предпоследний член
unsigned long last = 1;      // последний член
while(next < limit / 2 )     // результат не должен быть слишком большим
{
  cout << last << "  ";      // вывод последнего члена
  long sum = next + last;    // сложение двух последних членов
  next = last;               // обновление предпоследнего
  last = sum;                // и последнего членов
}
cout << endl;
return 0;
}
```

Вывод программы выглядит следующим образом:

```
1  2  3  5  8  13  21  34  55  89  144  233  377  610  987  1597  2584  4181  6765  10946
17711  28657  46368  75025  121393  196418  317811  514229  832040  1346269  2178309
3524578  5702887  9227465  14930352  24157817  39088169  63245986  102334155  165580141
267914296  433494437  701408733  1134903170  1836311903  2971215073
```

Отношение последних двух членов последовательности равно 0,618033988, что очень близко к «золотому сечению».

В программе FIBO используется тип unsigned long с максимальным среди целых типов диапазоном представления положительных чисел. Условие, проверяемое перед началом исполнения тела цикла, выводит программу из цикла до

того, как произойдет переполнение. Мы определяем величину, равную предельному значению переменной типа unsigned long, как константу, поскольку эта величина не меняется в ходе выполнения программы. Цикл должен прекратиться в том случае, если значение переменной next станет больше половины допустимой для данного типа границы. Значение переменной sum может достигать любых допустимых значений для данного типа. Условие продолжения цикла содержит две операции:

```
(next < limit / 2)
```

Нам необходимо сравнить значение next с limit/2, а значит, деление должно произойти перед сравнением. Для того чтобы быть уверенными в том, что произойдет именно это, мы можем использовать круглые скобки:

```
(next < (limit / 2))
```

На самом деле скобки не нужны. Почему? Потому что арифметические операции имеют более высокий приоритет, чем операции отношения. Таким образом, мы можем быть уверенными в том, что деление имеет более высокий приоритет, нежели операция «больше», и будет выполнено первым. Мы рассмотрим старшинство операций в полном объеме после того, как расскажем о логических операциях.

Цикл do

В цикле while условие продолжения выполнения цикла помещалось в начало цикла. Это означало, что в случае невыполнения условия при первой проверке тело цикла вообще не исполнялось. В некоторых случаях это целесообразно, но возможны и ситуации, когда необходимо выполнить тело цикла хотя бы один раз вне зависимости от истинности проверяемого условия. Для этого следует использовать цикл do, в котором условие продолжения цикла располагается не перед, а после тела цикла.

Наш следующий пример DIVDO выводит приглашение ввести два числа: делимое и делитель, а затем производит целочисленное деление с использованием операций / и % и выводит полученные частное и остаток.

```
// divdo.cpp
// применение цикла do
#include <iostream>
using namespace std;
int main()
{
  long dividend, divisor;
  char ch;
  do                      // начало цикла do
  {                       // действия
  cout << "Введите делимое: " ; cin >> dividend;
  cout << "Введите делитель: " ; cin >> divisor;
  cout << "Частное равно " << dividend / divisor;
  cout << ", остаток равен " << dividend % divisor;
```

```
  cout << "\nЕще раз?(y/n): ";
  cin >> ch;
  }
  while(ch != 'n'));    // условие цикла
  return 0;
}
```

Большая часть программы находится в составе тела цикла do. Ключевое слово do обозначает начало цикла. Затем, как и в других циклах, следует тело, обрамленное фигурными скобками. Завершает цикл условие продолжения, описываемое с помощью ключевого слова while. Это условие похоже на условие цикла while, но у него есть два отличия: оно располагается в конце цикла и завершается точкой с запятой (;). Синтаксис цикла do показан на рис. 3.5.

```
do ◯ ——Прим.: точка с запятой не нужна

statement; ———Тело цикла из одного оператора

while (ch! = 'n');
                  └— Прим.: точка с запятой
       └———— Условие выполнения


do ◯ ——Прим.: точка с запятой не нужна
   {
   statement;  } Тело цикла из нескольких
   statement;  } операторов — блок
   statement;
   }
while (numb < 96);
                  └— Прим.: точка с запятой
       └———— Условие выполнения
```

Рис. 3.5. Синтаксис цикла do

Перед тем как производить вычисление, программа DIVDO спрашивает пользователя, хочет ли он произвести это вычисление. Если в ответ программа получает символ 'y', то выражение

```
ch! = 'n'
```

сохраняет значение true. В случае, если пользователь вводит 'n', условие продолжения цикла не выполняется и происходит выход из цикла. На рис. 3.6 приведена схема функционирования цикла do.

Вот пример возможного вывода программы DIVDO:

```
Введите делимое: 11
Введите делитель: 3
Частное равно 3, остаток равен 2
Еще раз? (y/n): y
Введите делимое: 222
Введите делитель: 17
Частное равно 13, остаток равен 1
Еще раз? (y/n)· n
```

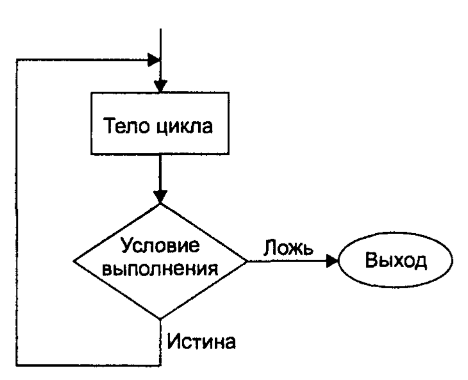


Рис. 3.6. Исполнение цикла do

Выбор типа цикла

Мы рассмотрели основные аспекты использования циклов. Цикл for подходит для тех случаев, когда мы заранее знаем, сколько раз нам потребуется его выполнение. Циклы while и for используются в тех случаях, когда число итераций цикла заранее не известно, причем цикл while подходит в тех случаях, когда тело цикла может быть не исполненным ни разу, а цикл do — когда обязательно хотя бы однократное исполнение тела цикла.

Эти критерии достаточно спорны, поскольку выбор типа цикла больше определяется стилем, нежели строго определенными правилами. Каждый из циклов можно применить практически в любой ситуации. При выборе типа цикла стоит руководствоваться удобочитаемостью и легкостью восприятия вашей программы.

Ветвления

Управление циклом всегда сводится к одному вопросу: продолжать выполнение цикла или нет? Разумеется, люди в реальной жизни встречаются с гораздо более разнообразными вопросами. Нам нужно решать не только, пойти ли на работу сегодня или нет (продолжить ли цикл), но и делать более сложный выбор, например, купить ли нам красную футболку или зеленую (или вообще не покупать футболку) или взять ли нам отпуск и, в случае положительного ответа, провести его в горах или на море?

Программам тоже необходимо принимать решения. В программе решение, или *ветвление*, сводится к переходу в другую часть программы в зависимости от значения соответствующего выражения.

В C++ существует несколько типов ветвлений, наиболее важным из которых является if...else, осуществляющее выбор между двумя альтернативами. В операторе ветвления if...else использование else не является обязательным. Для выбора одной из множества альтернатив используется оператор ветвления switch, действие которого определяется набором значений соответствующей перемен-

ной. Кроме того, существует так называемая *условная операция*, используемая в некоторых особых ситуациях. Мы рассмотрим каждую из этих конструкций.

Условный оператор if

Оператор if является наиболее простым из операторов ветвлений. Следующая программа, IFDEMO, иллюстрирует применение оператора if.

```
// ifdemo.cpp
// применение оператора if
#include <iostream>
using namespace std:
int main()
{
  int x:
  cout << "Введите число: ":
  cin >> x:
  if( x > 100 )
  cout << "Это число больше. чем 100\n ":
  return 0:
}
```

За ключевым словом if следует условие ветвления, заключенное в круглые скобки. Синтаксис оператора if показан на рис. 3.7. Легко заметить, что синтаксис if очень напоминает синтаксис while. Разница заключается лишь в том, что если проверяемое условие окажется истинным, то операторы, следующие за if, будут выполнены всего один раз; операторы, следующие за while, исполняются до тех пор, пока проверяемое условие не станет ложным. Функционирование оператора if показано на рис. 3.8.

Примером работы программы IFDEMO может служить следующий:

```
Введите число: 2000
Это число больше. чем 100
```

Если вводимое число окажется не превосходящим 100, то программа завершится, не напечатав вторую строку.

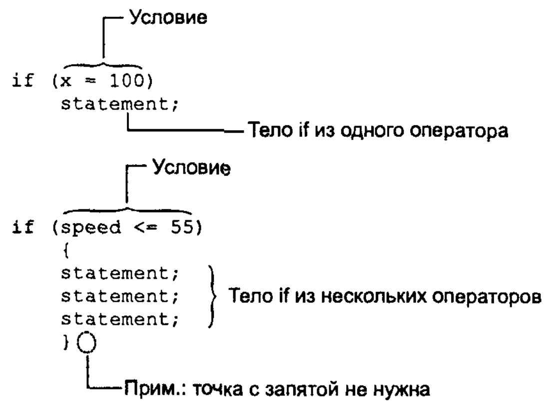


Рис. 3.7. Синтаксис оператора if

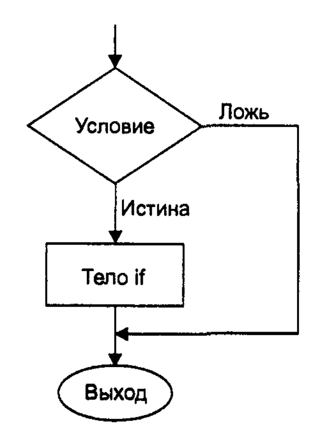


Рис. 3.8. Исполнение оператора if

Несколько операторов в теле if

Как и для циклов, тело ветвления if может состоять как из одного оператора, что было продемонстрировано в программе IFDEMO, так и из нескольких операторов, заключенных в фигурные скобки. Пример, иллюстрирующий использование блока операторов в теле if, называется IF2 и приводится ниже.

```
// if2.cpp
// использование нескольких операторов в теле цикла if
#include <iostream>
using namespace std;
int main()
{
  int x;
  cout << "Введите число: ";
  cin >> x;
  if( x > 100 )
  {
    cout << "Число " << x;
    cout << " больше, чем 100\n";
  }
  return 0;
}
```

Вот возможный результат работы программы IF2:

```
Введите число: 12345
Число 12345 больше, чем 100
```

if внутри циклов

Циклы и ветвления можно использовать совместно. Вы можете помещать ветвления внутрь цикла и наоборот, использовать вложенные ветвления и вложенные циклы. В следующем примере под названием PRIME ветвление if находится внутри цикла for. Программа определяет, является ли вводимое число простым

или нет (*простым* называется число, которое делится только на единицу и на само себя. Примерами простых чисел являются 2, 3, 5, 7, 11, 13 и 17).

```
// prime cpp
// применение цикла if для определения простых чисел
#include <iostream>
using namespace std;
#include <process.h>          //для exit()
int main()
{
  unsigned long n,j;
  cout << "Введите число: ";
  cin >> n;                   // ввод проверяемого числа
  for(j=2; j <= n/2; j++)     // деление на целые числа,
  if(n%j == 0)                // начиная с 2; если остаток
  {                           //нулевой, то число не простое
    cout << "Число не простое: делится на " << j << endl;
    exit(0);                  // выход из программы
  }
  cout << "Число является простым\n";
  return 0;
}
```

В этом примере пользователь вводит значение, которое присваивается переменной n. Затем программа при помощи цикла for делит число n на все числа от 2 до n/2. Делителем является переменная j, служащая счетчиком цикла. Если число n разделится без остатка на какое-либо из значений j, то оно не будет простым. Условием того, что одно число делится на другое без остатка, является равенство остатка от деления нулю. Поэтому в условии для if участвует операция остатка от деления %. Если число оказывается не простым, то мы выводим соответствующее сообщение и выходим из программы.

Ниже приведен результат работы программы для трех последовательно введенных чисел:

```
Введите число: 13
Число является простым
Введите число: 22229
Число является простым
Введите число: 22231
Число не простое: делится на 11
```

Обратите внимание — тело цикла не заключено в фигурные скобки. Это объясняется тем, что оператор if и операторы тела ветвления на самом деле являются одним оператором. Для того чтобы улучшить читаемость кода, вы можете добавить фигурные скобки, но это не является обязательным для правильной работы компилятора.

Функция exit()

Когда программа PRIME получает число, не являющееся простым, она завершается, поскольку нет необходимости несколько раз проверять, является число простым или нет. Библиотечная функция exit() производит немедленный выход из

программы независимо от того, в каком месте она находится. Эта функция не возвращает значения. Ее единственный аргумент (в нашем случае 0) возвращается вызывающему окружению после того, как программа завершается (эта величина часто используется в *пакетных* файлах, которые запрашивают значение, возвращаемое функцией exit(). Как правило, возвращаемое значение 0 говорит об успешном завершении программы; ненулевые значения сигнализируют об ошибках).

Оператор if...else

Оператор if позволяет совершать действие в том случае, если выполняется некоторое условие. Если же условие не выполняется, никакого действия не выполняется. Однако можно представить такую ситуацию, когда нам необходимо совершить одно действие в случае выполнения условия и другое действие в случае невыполнения этого условия. Здесь оказывается полезным ветвление if...else. Оно состоит из оператора if, за которым следует блок операторов, и ключевого слова else, за которым следует еще один блок операторов. Синтаксис ветвления показан на рис. 3.9.

```
         ┌─ Условие
if (x > 100)
    statement; ——Тело if из одного оператора
else
    statement; ——Тело else из одного оператора

            ┌─ Условие
if (zebra! = 0)
    {
    statement; } Тело if из нескольких операторов
    statement; }
    }
else
    {
    statement; } Тело else из нескольких операторов
    statement; }
    }
```

Рис. 3.9. Синтаксис if...else

Изменим программу if, добавив к ветвлению else-часть:

```
// ifelse.cpp
// применение конструкции if...else
#include <iostream>
using namespace std;
int main()
{
  int x;
  cout << "\nВведите число: ";
  cin >> x;
  if( x > 100 )
```

```
    cout << "Это число больше, чем 100\n";
  else
    cout << "Это число не больше, чем 100\n";
  return 0;
}
```

В зависимости от истинности или ложности условия ветвления, программа выводит на экран соответствующее сообщение. Вот результаты двух вызовов программы:

```
Введите число: 300
Это число больше, чем 100
Введите число: 3
Это число не больше, чем 100
```

Функционирование ветвления if...else показано на рис. 3.10.

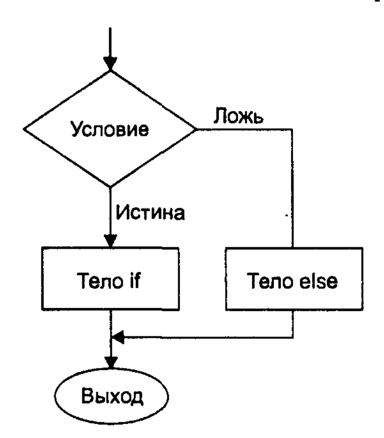


Рис. 3.10. Исполнение оператора if...else

Функция getche()

Следующий наш пример CHCOUNT демонстрирует использование ветвления if...else внутри цикла while. Кроме того, в нем используется библиотечная функция getche(). Программа подсчитывает количество слов и символов в строке, вводимой пользователем с клавиатуры.

```
// chcount.cpp
// подсчет числа слов и символов в строке
#include <iostream>
using namespace std;
#include <conio.h>          // для getche()
int main()
{
  int chcount = 0;          // число непробельных символов
  int wdcount = 1;          // число пробелов
  char ch = 'a';            // ch должна иметь определенное значение
  cout << "Введите строку: ";
  while(ch != '\r' ))       // цикл, пока не будет нажата клавиша Enter
  {
```

```
    ch = getche();              // считывание символа
    if( ch == ' ' )             // если символ является пробелом,
      wdcount++;                // то инкрементируем число слов
    else                        // в противном случае
      chcount++;                // инкрементируем число символов
  }                             // вывод результатов на экран
  cout << "\nСлов: " << wdcount << endl;
  << "Букв: " << (chcount-1)<< endl;
  return 0;
}
```

До сих пор мы использовали для ввода только объект cin и операцию >>. Такой способ вводить значения предполагает, что после ввода значения пользователь нажмет клавишу Enter. Это верно и в отношении отдельных символов: пользователь вводит символ, а затем нажимает Enter. В данном случае программе необходимо обрабатывать каждый введенный символ сразу после его появления, не дожидаясь нажатия клавиши Enter. Такую возможность обеспечивает библиотечная функция getche(). Эта функция не имеет аргументов, а ее описание содержится в заголовочном файле CONIO.H. Значение, возвращаемое функцией getche(), присваивается переменной ch (функция getche(), кроме возвращения значения, печатает это значение на экране; такой ввод называется ввод с эхом (echo), что отражено в названии функции буквой e на конце. Другая функция под названием getch() похожа на функцию getche(), но, в отличие от нее, не отображает вводимый символ на экране.

Ветвление if...else увеличивает на единицу значение переменной wdcount в том случае, если с клавиатуры вводится пробел, и значение переменной chcount, если введен не пробел. Таким образом, программа считает за букву любой символ, не являющийся пробелом. Разумеется, этот алгоритм примитивен и дает ошибочный результат уже при введении нескольких пробелов между словами.

Взаимодействие с программой может осуществляться следующим образом:

```
Введите строку: For while and do
Слов: 4
Букв: 13
```

Условие цикла while проверяет, не является ли нажатая клавиша клавишей Enter, что соответствует выдаче функцией getche() символа, соответствующего управляющей последовательности '\r'. При нажатии клавиши Enter цикл и программа завершаются.

Условия с присваиванием

Можно переписать программу CHCOUNT, сократив ее на одну строку и продемонстрировав важные аспекты, касающиеся присваивания и старшинства операций. Получившаяся в результате конструкция может показаться необычной, однако на самом деле она часто употребляется не только в C++, но даже в C.

Приведем измененную версию предыдущей программы, названную CHCNT2:

```
// chcnt2.cpp
// подсчет числа слов и символов в строке
```

```
#include <iostream>
using namespace std:
#include <conio.h>          // для getche()
int main()
{
  int chcount = 0:
  int wdcount = 1:          // пробел между двумя словами
  char ch:
  while( (ch=getche()) != '\r ' ) // цикл. пока не нажата клавиша Enter
  {
    if (ch == ' ')          // если введен пробел.
    wdcount++:              // инкрементировать счетчик слов
    else                    // иначе
    chcount++:              // инкрементировать число символов
  }                         // вывод результатов
  cout << "\nСлов: " << wdcount << endl
  <<"Букв: " << chcount << endl:
  return 0:
}
```

Значение, возвращаемое функцией getche(), присваивается переменной ch, как и раньше, но сама операция присваивания находится прямо внутри условия цикла while. Присвоенное значение сравнивается с '\r' для того, чтобы выяснить, продолжать выполнение цикла или нет. Эта конструкция работает правильно, потому что операция присваивания значения сама получает это значение. Если, к примеру, функция getche() возвращает символ 'a', то при этом не только происходит присваивание значения 'a' переменной ch, но само выражение

```
(ch = getche())
```

получает значение, равное 'a', которое участвует в проверке.

Тот факт, что операции присваивания имеют собственное значение, применяется при множественном присваивании, таком, как

```
x = y = z = 0:
```

Подобные конструкции являются абсолютно корректными в C++. Сначала переменной z присваивается значение 0, которое затем присваивается переменной y. После этого выражение y=z=0 получает значение 0, которое присваивается переменной x.

Внешние круглые скобки в выражении

```
(ch=getche())
```

необходимы, поскольку операция присваивания = имеет более низкий приоритет, чем операция отношения !=. Если бы скобки отсутствовали, то выражение

```
while (ch = (getche() != '\r'))
```

присваивало бы истинное или ложное значение переменной ch, что было бы неправильно для нашего алгоритма.

Таким образом, оператор while в программе CHCNT2 выполняет много полезных действий. Он не только проверяет, является ли значение переменной ch символом '\r', но еще получает символ с клавиатуры и присваивает его переменной ch. Непросто с одного взгляда понять все действия этого оператора.

Вложенные ветвления if...else

Возможно, вам приходилось видеть приключенческие игры, предназначенные для ранних версий MS DOS. Их суть заключалась в следующем: играющий двигал своего «героя» по воображаемому ландшафту и замкам среди волшебников, сокровищ и т. д., нарисованных с помощью текстовых символов. Следующая программа, ADIFELSE, напоминает небольшую часть такой приключенческой игры.

```
// adifelse.cpp
// приключенческая игра с применением ветвления if...else
#include <iostream>
using namespace std;
#include <conio.h>            // для getche()
int main()
{
  char dir='a';
  int x=10, y=10;
  cout << "Нажмите Enter для выхода...\n";
  while ( dir != '\r' )       // пока не будет нажата клавиша Enter
  {
    cout << "\nВаши координаты: " << x << ", " << y;
    cout << "\nВыберите направление (n,s,e,w): ";
    dir = getche();           // ввод символа
    if ( dir == 'n' )         // движение на север
      y--;
    else
      if ( dir == 's' )       // движение на юг
        y++;
      else
        if ( dir == 'e' )     // движение на восток
          x++;
        else
          if ( dir == 'w' )   // движение на запад
            x--;
  }                           // конец цикла while
  return 0;
}                             // конец функции main()
```

Когда игра начинается, вы оказываетесь на бесплодном участке земли. Вы можете передвигаться на север, юг, запад и восток, а программа будет следить за вашими передвижениями и сообщать ваши текущие координаты. Начало движения находится в точке с координатами (10, 10). С вашим героем не будет происходить ничего интересного, куда бы он ни пошел; пустая земля простирается во всех направлениях, как видно на рис. 3.11. Позже мы внесем в эту игру немного разнообразия.

Вот пример взаимодействия с нашей игрой:

```
Ваши координаты: 10, 10
Выберите направление (n,s,e,w): n
Ваши координаты: 10, 9
Выберите направление (n,s,e,w): e
Ваши координаты: 11, 9
Выберите направление (n,s,e,w):
```

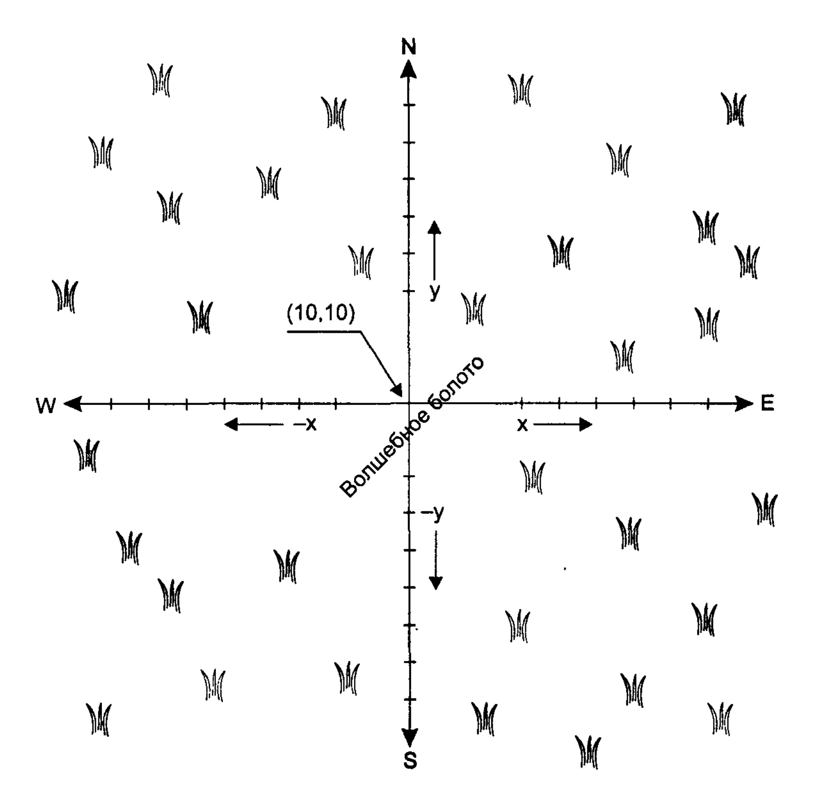


Рис. 3.11. Волшебное болото

Для выхода из программы нужно нажать клавишу Enter.

Данная программа не является шедевром среди видеоигр, однако в ней демонстрируется применение вложенных ветвлений. Так, оператор if находится внутри оператора if...else, который, в свою очередь, также является частью ветвления if...else. Если первое условие не выполняется, то проверяется второе условие и т. д. до тех пор, пока не будут проверены все условия. Если какое-либо из условий выполняется, то изменяется соответствующая координата x или y, после чего программа выходит из всех вложенных ветвлений. Подобные вложенные группы ветвлений называются *деревом ветвлений.*

if и else во вложенных ветвлениях

Во вложенных ветвлениях if...else возникает сложность установления соответствий между else и if. Часто можно перепутать, к какому из операторов if относится данный блок else. Программа BADELSE является примером, содержащим типичную ошибку, допускаемую при вложении ветвлений if...else.

```
// badelse.cpp
// неправильное сопоставление else и if во вложенных ветвлениях
#include <iostream>
```

```
using namespace std;
int main()
{
  int a, b, c;
  cout << "Введите числа a, b и c: \n";
  cin >> a >> b >> c;
  if ( a == b )
    if( b == c )
      cout << "a, b, и c равны\n";
  else
    cout << "a и b не равны\n";
  return 0;
}
```

Мы использовали ввод нескольких значений с помощью одного оператора с участием cin. Трижды вводя значения и нажимая клавишу Enter, вы инициализируете переменные a, b и c.

Что будет происходить в том случае, если вы последовательно введете значения 2, 3 и 3? Значения переменных a и b не равны, поэтому первое условие не будет выполнено и вы, вероятно, ожидаете, что программа перейдет к исполнению инструкций else и напечатает сообщение "a и b не равны". Но на самом деле ничего не печатается. Почему? Потому, что вы связываете else не с тем из операторов if. Вы, скорее всего, думаете, что else относится к первому из if-ветвлений, но на самом деле он связан со вторым. Общее правило выглядит так: else связан с последним из операторов if, который не имеет своего собственного блока else. Правильное использование if и else будет выглядеть так:

```
if ( a == b )
  if ( b == c )
    cout << "a, b, и c равны\n";
  else
    cout << "b и c не равны\n";
```

Мы изменили выравнивание и фразу, выводимую в теле else. Теперь в случае, если будут введены значения 2, 3 и 3, по-прежнему ничего не будет печататься, но если ввести значения 2, 2 и 3, то на экране появится фраза

```
b и c не равны
```

Если вам необходимо связать else с тем if, который расположен раньше, нужно использовать фигурные скобки:

```
if ( a == b )
{
  if ( b == c )
  cout << "a, b и c равны";
}
else
  cout << "a и b не равны";
```

Здесь else связан с первым оператором if, поскольку скобки делают второй по счету if невидимым для блока else.

Конструкция else...if

Вложенные ветвления if...else в программе ADIFELSE выглядят несколько неуклюже и могут представлять трудность для восприятия, особенно если глубина вложенности больше, чем показано в примере. Существует еще один вариант записи этих же действий. Необходимо только немного изменить последовательность записи кода программы. В результате получим следующий пример ADELSEIF:

```
// adelseif.cpp
// приключенческая игра с применением else ..if
#include <iostream>
using namespace std;
#include <conio.h>              // для getche()
int main()
{
  char dir='a';
  int x=10,y=10;
  cout << "Нажмите Enter для выхода...\n";
  while( dir != '\r' )          // пока не нажата клавиша Enter
  {
    cout << "\nВаши координаты: " << x << ", " << y;
    cout << "\nВыберите направление (n,s,e,w): ";
    dir = getche();             // ввод символа
    if ( dir == 'n' )           // движение на север
      y--;
    else if(dir=='s' ))         // движение на юг
      y++;
    else if(dir=='e' ))         // движение на восток
      x++;
    else if(dir=='w' ))         // движение на запад
      x--;
  }                             // конец цикла while
  return 0;
}                               // конец функции main()
```

Компилятор воспринимает эту программу как идентичную программе ADIFELSE, но здесь мы поменяли местами блоки if и else в ветвлениях. У вас может создаться впечатление, что мы использовали новый вид ветвления else...if. Программа последовательно исполняет блоки else...if до тех пор, пока не выполнится хотя бы одно из проверяемых условий. Затем исполняется соответствующий оператор и производится выход из ветвлений. Такой способ представления вложенных ветвлений гораздо проще и удобнее для понимания, чем обычная последовательность конструкций if...else.

Оператор switch

Если в вашей программе присутствует большое дерево ветвлений и все ветвления зависят от значения какой-либо одной переменной, то можно вместо ступен-

чатой последовательности конструкций if...else или else...if воспользоваться оператором switch. Рассмотрим простой пример под названием PLATTERS:

```
// platters.cpp
// применение ветвления switch
#include <iostream>
using namespace std:
int main()
{
  int speed:             // скорость вращения грампластинки
  cout << "\nВведите число 33,45 или 78: ";
  cin >> speed:          // ввод скорости пользователем
  switch(speed)          // действия, зависящие от выбора скорости
  {
    case 33:             // если пользователь ввел 33
      cout << "Долгоиграющий формат\n":
      break:
    case 45:             // если пользователь ввел 45
      cout << "Формат сингла\n":
      break:
    case 78:             // если пользователь ввел 78
      cout << "Устаревший формат\n":
      break:
  }
  return 0:
}
```

Эта программа печатает одно из трех сообщений в зависимости от того, какое из чисел — 33, 45 или 78 — введет пользователь. Как наверняка знают люди старшего поколения, долгоиграющие пластинки (LP), содержащие большое количество песен, имели скорость проигрывания, равную 33 оборотам в минуту; пластинки меньшего размера, рассчитанные на 45 оборотов в минуту, содержали по одной песне на каждой стороне; наконец, пластинки со скоростью 78 оборотов в минуту являлись предшественницами пластинок двух предыдущих форматов.

За ключевым словом switch следуют круглые скобки, содержащие имя переменной, от значений которой зависит дальнейшее выполнение программы:

```
switch(speed)
```

Скобки ограничивают набор операторов case. Каждый раз за ключевым словом case следует константа, после значения которой стоит двоеточие:

```
case 33:
```

Тип констант, употребляемых в операторах case, должен совпадать с типом переменной, стоящей в скобках оператора switch. На рис. 3.12 показан синтаксис оператора switch.

Перед входом в тело switch программа должна инициализировать каким-либо значением переменную, стоящую внутри оператора switch, поскольку это значение будет сравниваться с константами, стоящими в теле ветвления switch. Если какое-либо из сравнений даст истинный результат, то операторы, стоящие за соответствующим сравнением, будут исполняться до тех пор, пока не встретится слово break.

Вот пример работы программы PLATTERS:

```
Введите 33, 45 или 78: 45
Формат сингла
```

Оператор break

В конце последовательности операторов каждой из case-секций помещен оператор break. Этот оператор завершает выполнение ветвления switch. Управление в этом случае передается первому оператору, следующему за конструкцией switch. В примере PLATTERS таким оператором является завершение программы. Не забывайте ставить оператор break, поскольку при его отсутствии управление будет передано операторам, относящимся к другим веткам switch, что, скорее всего, создаст нежелательный эффект для работы программы (хотя в отдельных случаях это может оказаться полезным).

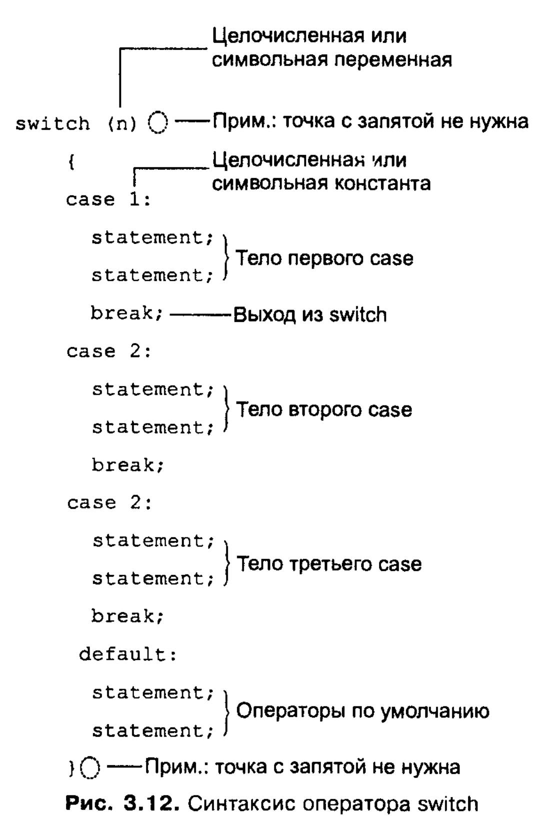


Рис. 3.12. Синтаксис оператора switch

Если значение переменной в операторе switch не совпадет ни с одним из значений констант, указанных внутри ветвления, то управление будет передано в конец switch без выполнения каких-либо других действий. Механизм функционирования конструкции switch приведен на рис. 3.13. Ключевое слово break

также используется для преждевременного выхода из циклов; скоро мы рассмотрим этот вопрос.

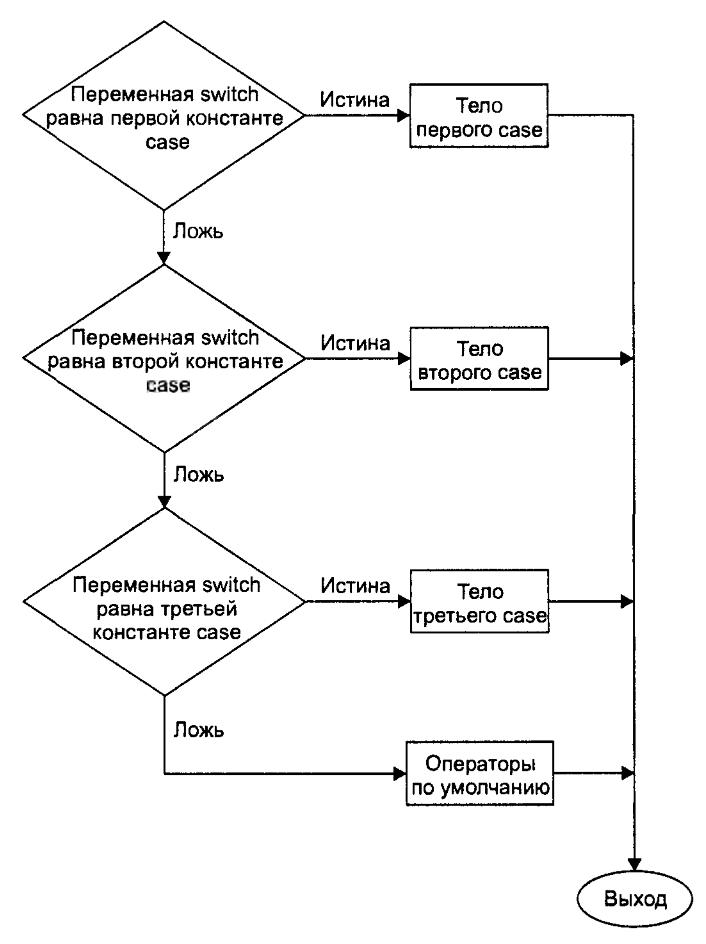


Рис. 3.13. Исполнение конструкции switch

switch и символьные переменные

Пример PLATTERS показывает механизм работы конструкции switch с использованием переменной типа int. Вместо типа int можно использовать также и тип char. Изменим программу ADELSEIF и назовем результат ADSWITCH:

```
// adswitch.cpp
// приключенческая игра с использованием switch
#include <iostream>
using namespace std;
#include <conio.h>          // для getche()
int main()
{
```

```
  char dir='a';
  int x=10,y=10;
  while( dir !='\r' )
  {
    cout << "\nВаши координаты: " << x << ", " << y;
    cout << "\nВыберите направление (n,s,e,w): ";
    dir = getche();          // ввод переменной
    switch(dir)              // switch с переменнной dir
    {
      case 'n': y--; break;  // движение на север
      case 's': y++; break;  // движение на юг
      case 'e': x++; break;  // движение на восток
      case 'w': x--; break;  // движение на запад
      case '\r': cout << "Выход...\n" ; break; // нажатие Enter
      default: cout << "Попробуйте еще\n";     // нажатие других клавиш
    }                        // конец switch
  }                          // конец while
  return 0;
}                            // конец main()
```

Символьная переменная dir используется в качестве «переключателя» в операторе switch, а константы 'n', 's' и т. д. служат значениями для case-конструкций (обратите внимание на то, что вы можете использовать в switch только целые и символьные типы, но не можете использовать вещественные).

Поскольку case-секции очень невелики по своему размеру, мы записали все операторы каждой секции в одной строке. Кроме того, мы добавили ветвь, выводящую сообщение о выходе в случае нажатия клавиши Enter.

Ключевое слово default

Последняя ветвь switch программы ADSWITCH начинается не со слова case, а с нового ключевого слова default. Оно предназначено для того, чтобы программа могла выполнить некоторую последовательность действий в том случае, если ни одно из значений констант не совпало со значением switch-переменной. В последнем примере мы выводим сообщение Попробуйте еще в том случае, если пользователь ввел некорректный символ. Поскольку секция default находится последней в теле switch, для выхода из ветвления не нужно указывать ключевое слово break.

Ветвление switch часто используется для работы со значением, которое вводится пользователем. Множество возможных таких значений отражено в операторах case. Использование оператора default может быть очень полезным даже в тех случаях, когда, на первый взгляд, нет необходимости в его применении. Например, конструкция

```
default:
 cout << "Ошибка: некорректное значение"; break;
```

может сигнализировать о том, что программа функционирует неправильно.

Из соображений краткости мы иногда будем опускать конструкцию default в других наших программах, но вам рекомендуется использовать ее, особенно в сложных программах.

Сравнение switch и if...else

Когда лучше использовать последовательность if...else (или else...if), а когда — switch? При использовании else...if можно проверять значения разных не связанных друг с другом переменных, причем сравнения могут быть любой степени сложности:

```
if ( SteamPressure*Factor > 56 )
  // операторы
else if ( VoltageIn+VoltageOut < 23000)
  // операторы
else if ( day == Thursday )
  // операторы
else
  // операторы
```

В операторе switch все ветвления используют одну и ту же переменную; единственная возможность проверки в таком ветвлении — сравнение значения переменной с заданной константой. В частности, невозможно выполнить такую проверку:

```
case a<3:
  //действия
  break;
```

Совместно с case могут использоваться константы, имеющие либо целый, либо символьный тип, например 3 или 'a', или же константные выражения, приводящие к вычислению значения одного из указанных типов, например 'a'+32.

Если хорошо усвоить эти условия, написание конструкций switch становится простым и доступным для понимания. switch рекомендуется использовать там, где это возможно, особенно в случаях, когда дерево ветвлений содержит много ветвей.

Условная операция

В этом разделе пойдет разговор об операции, выполняющей несколько нетипичные действия. Существует распространенная в программировании ситуация: переменной необходимо присвоить одно значение в случае выполнения некоторого условия и другое значение в случае невыполнения этого условия. В следующем фрагменте переменной min присваивается наименьшее из значений alpha и beta с помощью конструкции if...else:

```
if ( alpha < beta )
  min = alpha;
else
  min = beta;
```

Подобные действия на практике оказались настолько распространенными, что была специально разработана *условная операция*, выполняющая эти действия. Эта операция записывается с помощью двух знаков и использует три опе-

ранда. Она является единственной операцией в C++, использующей более двух операндов. С помощью условной операции можно записать предыдущий фрагмент следующим образом:

```
min = ( alpha < beta ) ? alpha : beta;
```

Правая часть оператора представляет собой условное выражение:

```
( alpha < beta ) ? alpha : beta    // условное выражение
```

Знак вопроса ? и двоеточие : обозначают условную операцию. Условие, стоящее перед знаком вопроса

```
( alpha < beta )
```

является условием проверки. Это условие, вместе с переменными alpha и beta, составляет тройку операндов условной операции.

Если значение проверяемого условия истинно, то условное выражение становится равным значению alpha; в противном случае оно становится равным beta. Скобки вокруг проверяемого условия не обязательны, но их довольно часто употребляют для того, чтобы упростить визуальное восприятие этого оператора. На рис. 3.14 показан синтаксис оператора условия, а на рис. 3.15 — механизм его действия.

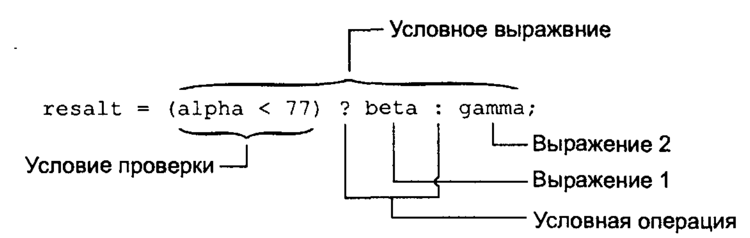


Рис. 3.14. Синтаксис условного оператора

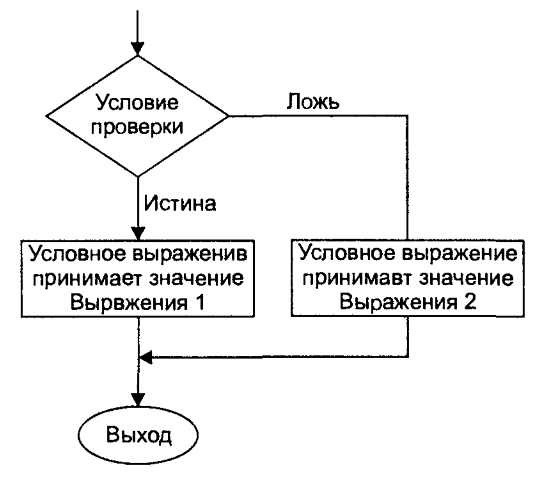


Рис. 3.15. Исполнение условного оператора

Значение условного выражения можно присваивать другим переменным и использовать его в любом месте, где допускается использование переменной. В данном примере условное выражение используется с переменной min.

Приведем пример использования условной операции для нахождения абсолютной величины (модуля) числа.

```
absvalue = n < 0 ? -n : n;
```

Если значение n окажется меньше нуля, то условное выражение принимает значение, противоположное n; в противном случае оно становится равным n. Полученное значение присваивается переменной **absvalue**.

Следующая программа, CONDI.CPP, использует условную операцию для вывода последовательности символов 'x', разделенных 8 пробелами.

```
// condi.cpp
// печать символа 'x' каждые 8 позиций
// с применением условной операции
#include <iostream>
using namespace std;
int main()
{
  for ( int j=0; j<80 ; j++)      // для каждой позиции
  {                               //значение ch равно 'x', если
    char ch = (j%8) ? ' ' :: 'x'; // номер позиции кратен 8,
    cout <<ch;                    // в противном случае ch содержит пробел
  }
  return 0;
}
```

Правая часть вывода частично не видна на экране из-за его ширины, но весь вывод выглядит однородно:

```
xxxxxxxxxx
```

Переменная j в цикле пробегает значения от 0 до 79, а операция остатка от деления j на 8, использующаяся как часть условия, принимает нулевое значение только тогда, когда j кратно 8. Таким образом, условное выражение

```
(j%8) ? ' ' : 'x'
```

принимает значение ' ' в тех случаях, когда j не кратно 8, и значение 'x', когда j кратно 8.

Кажется, что такая форма записи уже максимально сократила размер кода, однако это не так. Мы можем исключить из кода переменную ch, соединив два оператора в один следующим образом:

```
cout << ( (j%8) ? ' ' : 'x');
```

Некоторые «горячие головы» любят уплотнять код настолько, насколько это возможно. Однако делать это совсем не обязательно, и зачастую максимальное сокращение размера кода не стоит усилий, прилагаемых программистом к его сжатию. Даже использование условной операции зависит только от вашего желания: конструкция if...else с добавлением небольшого числа строк кода выполняет эквивалентную последовательность действий.

Логические операции

До сих пор мы использовали только два типа операций (если не считать условную операцию): арифметические операции (+, -, *, / и %) и операции отношения (<, >, <=, >=, == и !=). Теперь мы рассмотрим третью группу операций, называемых *логическими* операциями. Эти операции позволяют производить действия над булевыми переменными, то есть переменными, обладающими только двумя значениями — истина и ложь. Так, например, высказывание «Сегодня рабочий день» имеет булево значение, поскольку оно может быть только истинным или ложным. Другим примером булева выражения является высказывание «Мария поехала на автомобиле». Мы можем логически связать эти два высказывания: «если сегодня рабочий день и Мария поехала на автомобиле, то я должен воспользоваться автобусом». Логическая связь в данном примере устанавливается с помощью соединительного союза *и*, который определяет истинность или ложность комбинации двух соединяемых высказываний. Только в том случае, если оба высказывания являются истинными, я буду вынужден воспользоваться автобусом.

Операция логического И

Давайте рассмотрим, каким образом логические операции соединяют булевы выражения в C++. В следующем примере, ADVENAND, логическая операция используется для усложнения приключенческой игры, созданной в программе ADSWITCH. Теперь мы закопаем сокровище в точке с координатами (7, 11) и попробуем заставить нашего героя отыскать его.

```
// advenand.cpp
// применение операции логического И
#include <iostream>
using namespace std;
#include <process.h>          // для exit()
#include <conio.h>            // для getche()
int main()
{
  char dir='a';
  int x=10, y=10;
  while ( dir != '\r' )
  {
    cout << "\nВаши координаты: " << x << ", " << y;
    cout << "\nВыберите направление (n,s,e,w): ";
    dir = getche();           // ввод направления
    switch(dir)
    {
      case 'n': y--; break;   // обновление координат
      case 's': y++; break;
      case 'e': x++; break;
      case 'w': x--; break;
    }
    if( x==7 && y==11 )       // если x равно 7 и y равно 11
    {
      cout <<"\nВы нашли сокровище!\n";
      exit(0);                // выход из программы
```

```
    }
  }                          //конец switch
  return 0;
}                            //конец main()
```

Ключевым моментом данной программы является выражение

```
if( x == 7 && y == 11)
```

Условие, участвующее в этом выражении, будет истинным только в том случае, когда значение x будет равно 7, а значение y в это же время окажется равным 11. Операция *логического И*, обозначаемая &&, связывает пару *относительных* выражений (под относительным выражением понимается выражение, содержащее операции отношения).

Обратите внимание на то, что использование скобок, заключающих относительные выражения, не является обязательным:

```
( (x == 7 && y == 11) )      // внутренние скобки не обязательны
```

Это объясняется тем, что операции отношения имеют более высокий приоритет, чем логические операции.

Если игрок попадет в точку, где находится сокровище, программа отреагирует на это следующим образом:

```
Ваши координаты: 7, 10
Выберите направление (n, s, e, w): s
Вы нашли сокровище!
```

В C++ существуют 3 логические операции:

Операция	Название
&&	Логическое И
||	Логическое ИЛИ
!	Логическое НЕ

Операция «исключающее ИЛИ» в языке C++ отсутствует.

Теперь обратимся к примерам использования операций || и !.

Логическое ИЛИ

Введем в нашу приключенческую игру новых персонажей — драконов, которые будут обитать на западных и восточных землях и ограничат свободное передвижение нашего героя. Следующая программа, ADNENOR, с помощью операции логического ИЛИ реализует нашу задумку:

```
// advenor.cpp
// применение операции логического ИЛИ
#include <iostream>
using namespace std;
#include <process.h>         // для exit()
#include <conio.h>           // для getche()
int main()
{
```

```
    char dir='a';
    int x=10, y=10;
    while( dir != '\r' )        //выход при нажатии Enter
    {
      cout << "\n\nВаши координаты: " << x << ", " << y;
      if( x<5 || x>15 )         //если x меньше 5 или больше 15
      cout << "\nОсторожно - драконы!";
      cout <<"\nВыберите направление (n,s,e,w): ";
      dir = getche();           //выбор направления
      switch(dir)
      {
        case 'n': y--; break;   // обновление координат
        case 's': y++; break;
        case 'e': x++; break;
        case 'w': x--; break;
      }                         // конец switch
    }                           // конец while
    return 0;
}                               // конец main()
```

Выражение

```
x<5 || x>15
```

является истинным как в случае, если x меньше 5 (герой находится на западных землях), так и в случае, если x больше 15 (герой находится на восточных землях). Аналогично предыдущей программе, логическая операция || имеет более низкий приоритет, чем операции отношения < и >, поэтому дополнительные скобки в выражении не требуются.

Логическое НЕ

Операция логического НЕ является унарной, то есть имеет только один операнд (почти все операции C++, которые мы рассматривали, являлись бинарными, то есть имели два операнда; условная операция служит примером тернарной операции, поскольку имеет три операнда). Действие операции ! заключается в том, что она меняет значение своего операнда на противоположное: если операнд имел истинное значение, то после применения операции ! он становится ложным, и наоборот.

Например, выражение (x==7) является истинным, если значение x равно 7, а выражение !(x==7) является истинным для всех значений x, которые не равны 7 (в данной ситуации последнее выражение эквивалентно записи x!=7).

Целые величины в качестве булевых

Из всего вышесказанного у вас могло сложиться впечатление, что для того, чтобы выражение имело истинное или ложное значение, необходимо, чтобы это выражение включало в себя операцию отношения. На самом деле любое выражение целого типа можно рассматривать как истинное или ложное, даже если это выражение является обычной переменной. Выражение x рассматривается как истинное в том случае, если его значение не равно нулю, и как ложное, если его значе-

ние равно нулю. Очевидно, что в этом случае выражение !x истинно, если x равен нулю, и ложно, если x не равен нулю.

Давайте применим описанную концепцию на практике. Пусть в нашей приключенческой игре появятся грибы, причем расти они будут в тех точках, где обе координаты, x и y, будут кратны 7 (в приключенческих играх грибы, найденные героем, придают ему различные сверхъестественные возможности). Как мы знаем, кратность x числу 7 равносильна тому, что остаток от деления x на 7 равен нулю. Таким образом, применяя операцию остатка от деления, можно записать условие, определяющее местоположение грибов:

```
if ( x%7==0 && y%7==0 )
 cout << "Здесь находится гриб.\n";
```

С учетом того, что выражения могут рассматриваться как истинные или ложные без применения операций отношения, мы можем записать условие, стоящее в первой строке, более кратко следующим образом:

```
if ( !(x%7) && !(y%7) )       // если x%7 равно 0 и y%7 равно 0
```

Как мы говорили, логические операции && и || имеют более низкий приоритет, чем операции отношения. В таком случае зачем мы ставим скобки вокруг x%7 и y%7? Мы делаем это потому, что, несмотря на принадлежность операции ! к логическому типу, эта операция является унарной и имеет более высокий приоритет, чем бинарные операции отношения.

Приоритеты операций C++

Давайте теперь сведем в единую таблицу приоритеты всех операций, которые мы рассмотрели. Приоритет операций в таблице убывает сверху вниз, то есть операции, занимающие верхние строки, имеют приоритет выше, чем операции, находящиеся в нижних строках таблицы. Операции, находящиеся в одной строке, имеют одинаковый приоритет. Чтобы повысить приоритет операции, можно заключать ее в круглые скобки.

Более подробная таблица приоритетов операций приведена в приложении Б «Таблица приоритетов операций C++ и список зарезервированных слов».

Тип операций	Операции	Приоритет
Унарные	!, ++, --, +, -	Высший
Арифметические	Мультипликативные *, / , %	
	Аддитивные +, -	
Отношения	Неравенства <,>,<=,=>	
	Равенства ==, !=	
Логические	И &&	
	ИЛИ ||	
Условная	?:	
Присваивания	=, +=, -=, *=, /=, %=	Низший

Если при составлении относительного выражения с несколькими операциями у вас возникают ошибки, то рекомендуется использовать круглые скобки даже там, где это не является обязательным. Скобки не оказывают нежелательного воздействия на выражение и гарантируют правильный порядок его вычисления даже в том случае, если вы не знаете приоритетов операций. Кроме того, наличие скобок делает смысл выражения более ясным для того, кто читает листинг вашей программы.

Другие операторы перехода

В языке C++ существует несколько операторов перехода. Один из них, break, мы уже использовали при создании ветвлений switch, но операторы перехода можно использовать и в других целях. Оператор continue используется в циклах, а третий оператор перехода, goto, вообще, по возможности, не рекомендуется использовать. Рассмотрим эти операторы подробнее.

Оператор break

Оператор break производит выход из цикла подобно тому, как он действовал в конструкции switch. Следующим оператором, исполняемым после break, будет являться первый оператор, находящийся вне данного цикла. Это проиллюстрировано на рис. 3.16.

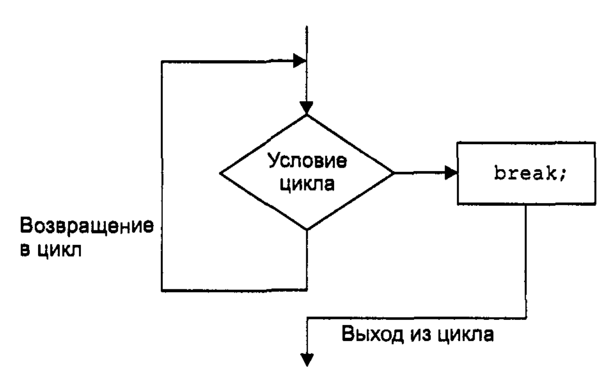


Рис. 3.16. Исполнение оператора break

Для того чтобы продемонстрировать использование оператора break, приведем программу, которая изображает распределение простых чисел:

```
// showprim.cpp
// изображает распределение простых чисел
#include <iostream>
using namespace std;
#include <conio.h>                  // для getche()
int main()
{
  const unsigned char WHITE =219; // белый цвет для простых чисел
```

```
  const unsigned char GRAY =176;    // серый цвет для остальных чисел
  unsigned char ch;
  // действия для каждой позиции на экране
  for(int count=0; count<80*25-1 ;count++)
  {
    ch = WHITE;                     //предполагаем, что число простое
    for(int j=2; j<count; j++)      // делим на каждое целое, большее 2
    if( count%j == 0)               // если остаток равен 0,
    {
      ch = GRAY;                    // то число не простое
      break;                        // выход из внутреннего цикла
    }
    cout << ch;                     // вывод символа на экран
  }
  getch();                          // задержка полученного изображения
  return 0;
}
```

Все позиции консоли размером 80×25 символов пронумерованы числами от 0 до 1999. Если число является простым, то позиция с соответствующим номером закрашивается белым цветом; в противном случае позиция закрашивается серым цветом.

На рисунке 3.17 представлен экран с результатом работы программы. Строго говоря, числа 0 и 1 не являются простыми, но соответствующие позиции закрашены белым, чтобы не вносить лишней путаницы в программу. Колонки, соответствующие одинаковым позициям в каждой строке, нумеруются с 0 по 79. В колонках с четными номерами не может находиться ни одно простое число, кроме 2, поскольку все четные числа делятся на 2. Есть ли закономерность распределения для остальных простых чисел? Пока математики не нашли такой формулы, которая позволяла бы определять, является ли число простым или нет.

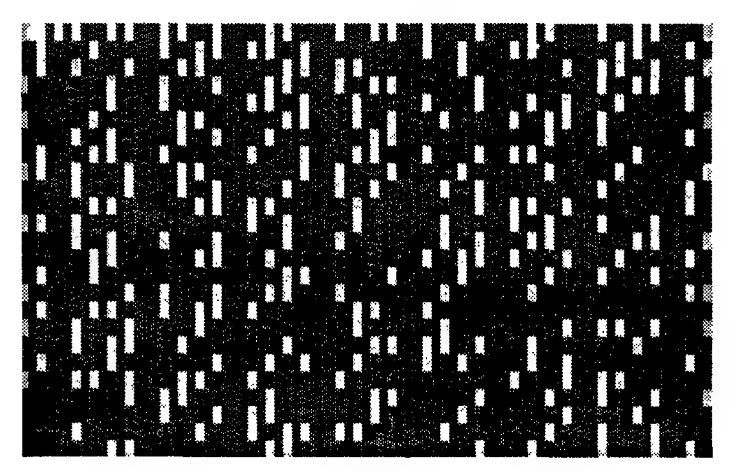


Рис. 3.17. Результат работы программs SHOWPRIM

Когда программа с помощью внутреннего цикла for определяет, что число не является простым, переменной ch присваивается значение GRAY, и программа выходит из внутреннего цикла с помощью оператора break (мы не хотим про-

изводить выход из программы, поскольку нам необходимо обработать не одну, а множество последовательностей чисел).

Обратите внимание на то, что оператор break производит выход только из одного цикла с максимальной глубиной вложения. Не имеет значения, какие конструкции и с какой глубиной вложены друг в друга: break производит выход только из одной такой конструкции, в которой он сам находится. Например, если бы внутри нашего цикла for находилось ветвление switch, то оператор break внутри switch вывел бы нас за пределы конструкции switch, но оставил бы внутри цикла for.

Последний оператор с cout печатает псевдографический символ, и затем цикл начинается для следующего числа.

Расширенная таблица символов ASCII

Эта программа использует два символа из расширенной таблицы символов ASCII, в которой представлены символы с кодами от 128 до 255 (см. приложение А «Таблица ASCII»). Число 219 соответствует закрашенному прямоугольнику (белого цвета на черно-белом мониторе), а число 176 — прямоугольнику серого цвета.

Программа SHOWPRIM использует функцию getch() в последней строке для того, чтобы после выполнения программы командная строка DOS появилась не сразу, а после нажатия любой клавиши. Таким образом, экран «застывает» до тех пор, пока клавиша не будет нажата.

Мы используем для символьных переменных тип unsigned char потому, что тип char способен хранить только числа, не превышающие 127, в то время как беззнаковая версия этого же типа позволяет хранить числа величиной до 255.

Оператор continue

Оператор break производит выход из цикла. Тем не менее, могут возникнуть и такие ситуации, когда необходимо при определенном условии не выходить из цикла, а досрочно возвращаться в его начало. Именно таким эффектом обладает применение оператора continue (строго говоря, continue делает переход на завершающую фигурную скобку цикла, откуда производится обычный переход в начало тела цикла). Действие оператора continue проиллюстрировано на рис. 3.18.

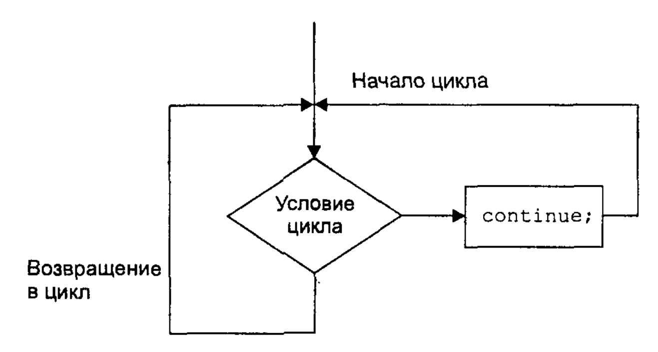


Рис. 3.18. Исполнение выражения continue

Теперь немного изменим программу DIVDO. Как мы уже видели, эта программа производит операцию деления, но в ней возможна фатальная ошибка: если пользователь введет ноль в качестве делителя, то произойдет аварийное завершение программы с выдачей сообщения Ошибка деления на ноль. Мы изменим исходный пример таким образом, чтобы подобная ситуация обрабатывалась более аккуратным способом.

```
// divdo2.cpp
// применение оператора continue
#include <iostream>
using namespace std;
int main()
{
  long dividend, divisor;
  char ch;
  do {
    cout << "Введите делимое: "; cin >> dividend;
    cout << "Введите делитель: " ; cin >> divisor;
    if( divisor == 0 )        // при попытке
    {                         // деления на ноль
      cout << "Некорректный делитель!\n"; // вывод сообщения
      continue;                           // возврат в начало цикла
    }
    cout << "Частное равно " << dividend / divisor;
    cout << ", остаток равен " << dividend % divisor;
    cout << "\nЕще раз?(y/n): ";
    cin >> ch;
  } while( ch != 'n' );
  return 0;
}
```

В том случае, если пользователь введет 0 в качестве делителя, программа выдаст сообщение о недопустимости введенного значения и повторит запросы на ввод значений сначала. Примером работы с такой программой является следующий:

```
Введите делимое: 10
Введите делитель: 0
Некорректный делитель!
Введите делимое:
```

Оператор break, будучи примененным в данной ситуации, выполнил бы выход одновременно из цикла do и из программы, что явилось бы не лучшим разрешением ситуации.

Обратите внимание на то, что мы немного сократили формат цикла do: ключевое слово do находится в одной строке с открывающей фигурной скобкой, а слово while — с закрывающей фигурной скобкой.

Оператор goto

Мы рассмотрим оператор goto в большей степени для того, чтобы дать исчерпывающие сведения об операторах переходов, нежели для того, чтобы вы использовали его в своих программах. Если вы хотя бы немного знакомы с принципами

построения программ, то вы знаете, что использование операторов goto способно легко запутать логику программы и сделать ее трудной для понимания и исправления ошибок. Практически не существует ситуаций, в которых использование оператора goto является необходимостью, и примеры, использованные в данной книге, лишний раз подтверждают это.

Синтаксис оператора goto следующий: вы вставляете метку перед тем оператором вашей программы, на который вы намечаете сделать переход. После метки всегда ставится двоеточие. Имя метки, перед которой расположено ключевое слово goto, совершит переход на тот оператор, который помечен данной меткой. Следующий фрагмент кода иллюстрирует применение goto:

```
goto System Crash;
// операторы
SystemCrash:
// сюда передается управление оператором goto
```

Резюме

Операции отношения предназначены для сравнения двух величин, то есть для проверки их на отношения «равенство», «больше», «меньше» и т. д. Результатом сравнения служит логическая, или булева, переменная, которая может иметь истинное или ложное значение. Ложное значение представляется нулем, истинное — единицей или любым другим числом, отличным от нуля.

В C++ существует 3 вида циклов. Цикл for чаще всего используется в тех случаях, когда число исполнений цикла известно заранее. Циклы while и do используются тогда, когда условие для завершения цикла формируется в процессе выполнения цикла, причем тело цикла while может не исполняться ни разу, а тело цикла do всегда исполняется хотя бы один раз. Тело цикла может представлять собой как один оператор, так и блок операторов, заключенный в фигурные скобки. Переменная, определенная внутри блока, доступна, или видима, только внутри этого блока.

Существует четыре вида ветвлений. Оператор if выполняет указанные в нем действия только тогда, когда выполняется некоторое условие. Оператор if...else выполняет одну последовательность действий в случае истинности проверяемого условия и другую последовательность действий в случае, если проверяемое условие ложно. Конструкция else if служит более наглядной формой последовательности вложенных друг в друга операторов if...else. Оператор switch организует множество ветвлений, зависящих от значения одной переменной. Условная операция возвращает одно из двух заданных значений в зависимости от истинности или ложности результата проверки соответствующего условия.

Операции логического И и логического ИЛИ предназначены для выполнения действий над булевыми переменными; результатом этих операций служит также булева переменная. Операция логического НЕ меняет значение своего операнда на противоположное: истину на ложь, и наоборот.

Оператор break передает управление первому оператору, следующему за циклом или ветвлением, в котором находится сам оператор break. Оператор continue

передает управление в начало того цикла, в котором он находится, а оператор goto — оператору, имеющему соответствующую метку.

Приоритеты операций определяют порядок вычисления значения выражения. Если расположить группы операций в порядке убывания их приоритета, то получим следующую последовательность: унарные, арифметические, отношения, логические, условная операция, операции присваивания.

Вопросы

Ответы на нижеприведенные вопросы можно найти в приложении Ж.

1. Операция отношения:
   а) присваивает значение одного операнда другому операнду;
   б) имеет своим результатом булево значение;
   в) сравнивает значения двух операндов;
   г) создает логическую комбинацию двух операндов.
2. Напишите выражение, использующее операцию отношения, результатом которого является истина, если значения переменных george и sally не равны.
3. Истинным или ложным является значение −1?
4. Назовите и опишите основное назначение каждого из трех выражений, входящих в состав оператора цикла for.
5. В цикле for, тело которого состоит более чем из одного оператора, точка с запятой ставится после:
   а) оператора цикла for;
   б) закрывающей фигурной скобки, ограничивающей тело цикла;
   в) каждого оператора в теле цикла;
   г) условия продолжения цикла.
6. Истинно ли следующее утверждение: инкрементирующее выражение цикла может декрементировать счетчик цикла?
7. Создайте цикл for, который будет выводить на экран числа от 100 до 110.
8. Блок кода ограничен ______.
9. Переменная, описанная внутри блока, видима:
   а) от точки своего объявления до конца программы;
   б) от точки своего объявления до конца функции;
   в) от точки своего объявления до конца блока;
   г) внутри функции.
10. Создайте цикл while, который будет выводить на экран числа от 100 до 110.
11. Истинно ли следующее утверждение: операции отношения имеют более высокий приоритет, чем арифметические операции?

12. Сколько раз исполняется тело цикла do?
13. Создайте цикл do, который будет выводить на экран числа от 100 до 110.
14. Напишите ветвление if, печатающее слово Yes в случае, если значение переменной age больше, чем 21.
15. Библиотечная функция exit() предназначена для выхода из:
    а) цикла, в котором она содержится;
    б) блока, в котором она содержится;
    в) функции, в которой она содержится;
    г) программы, в которой она содержится.
16. Напишите ветвление if...else, которое выводит на экран слово Yes, если значение переменной age больше, чем 21, и слово No в противном случае.
17. Библиотечная функция getche():
    а) возвращает символ в случае нажатия какой-либо из клавиш;
    б) возвращает символ в случае нажатия клавиши Enter;
    в) печатает на экране символ, соответствующий нажатой клавише;
    г) не отображает символ на экране.
18. Какой символ возвращается объектом cin в том случае, если пользователь нажимает клавишу Enter?
19. else всегда соответствует _______ if, если только if не ________.
20. Конструкция else...if получается из вложенных циклов if...else путем ___ ______.
21. Напишите ветвление switch, печатающее слово Yes в случае, если значение переменной ch равно 'y', No в случае, если ch равно 'n', и Unknown во всех остальных случаях.
22. Напишите оператор с участием условной операции, который присваивал бы переменной ticket значение, равное 1 в том случае, если значение переменной speed больше 55, и 0 в противном случае.
23. Операции && и ||:
    а) сравнивают два численных значения;
    б) комбинируют два численных значения;
    в) сравнивают два булевых значения;
    г) комбинируют два булевых значения.
24. Напишите выражение с участием логической операции, принимающее истинное значение, если значение переменной limit равно 55, а значение переменной speed превышает 55.
25. Перечислите в порядке убывания приоритетов следующие типы операций: логические, унарные, арифметические, присваивания, отношения, условная операция.

26. Оператор break производит выход:
    а) только из цикла наибольшей глубины вложенности;
    б) только из ветвления switch наибольшей глубины вложенности;
    в) из всех вложенных циклов и ветвлений;
    г) из цикла или ветвления наибольшей глубины вложенности.
27. Выполнение оператора continue внутри цикла приводит к передаче управления __________.
28. Оператор goto вызывает переход на:
    а) операцию;
    б) метку;
    в) переменную;
    г) функцию.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Предположим, вы хотите создать таблицу умножения на заданное число. Напишите программу, которая позволяет пользователю ввести это число, а затем генерирует таблицу размером 20 строк на 10 столбцов. Первые строки результата работы программы должны выглядеть примерно следующим образом:

```
Введите число: 7
7142128354249566370
77849198105112119126133140
147154161168175182189196203210
```

*2. Напишите программу, предлагающую пользователю осуществить перевод температуры из шкалы Цельсия в шкалу Фаренгейта или наоборот, а затем осуществите преобразование. Используйте в программе переменные вещественного типа. Взаимодействие программы с пользователем может выглядеть следующим образом:

```
Нажмите 1 для перевода шкалы Цельсия в шкалу Фаренгейта,
2 для перевода шкалы Фаренгейта в шкалу Цельсия: 1
Введите температуру по Фаренгейту: 70
Значение по Цельсию: 21.111111
```

*3. Операции ввода, такие, как cin, должны уметь преобразовывать последовательность символов в число. Напишите программу, которая позволяет пользователю ввести шесть цифр, а затем выводит результат типа long на экране. Каждая цифра должна считываться отдельно при помощи функции getche(). Вычисление значения переменной производится путем ум-

ножения текущего ее значения на 10 и сложения с последней введенной цифрой (для того, чтобы из кода символа получить цифру, вычтите из него 48 или '0'). Примером результата работы программы может служить следующий:

```
Введите число: 123456
Вы ввели число 123456
```

*4. Создайте эквивалент калькулятора, выполняющего четыре основных арифметических операции. Программа должна запрашивать ввод пользователем первого операнда, знака операции и второго операнда. Для хранения операндов следует использовать переменные вещественного типа. Выбрать операцию можно при помощи оператора switch. В конце программа должна отображать результат на экране. Результат работы программы с пользователем может выглядеть следующим образом:

```
Введите первый операнд, операцию и второй операнд: 10 / 3
Результат равен 3.333333
Выполнить еще одну операцию (y/n)? y
Введите первый операнд, операцию и второй операнд: 12 + 100
Результат равен 112
Выполнить еще одну операцию (y/n)? n
```

5. При помощи цикла for изобразите на экране пирамиду из символов 'X'. Верхняя часть пирамиды должна выглядеть следующим образом:

```
    X
   XXX
  XXXXX
 XXXXXXX
XXXXXXXXX
```

Вся пирамида должна быть высотой не 5 линий, как изображено здесь, а 20 линий. Одним из способов ее построения может служить использование двух вложенных циклов, из которых внутренний будет заниматься печатанием символов 'X' и пробелов, а другой осуществлять переход на одну строку вниз.

6. Измените программу factor, приведенную в этой главе, таким образом, чтобы она циклически запрашивала ввод пользователем числа и вычисляла его факториал, пока пользователь не введет 0. В этом случае программа должна завершиться. При необходимости вы можете использовать соответствующие операторы программы factor в цикле do или while.

7. Напишите программу, рассчитывающую сумму денег, которые вы получите при вложении начальной суммы с фиксированной процентной ставкой дохода через определенное количество лет. Пользователь должен вводить с клавиатуры начальный вклад, число лет и процентную ставку. Примером результата работы программы может быть следующий:

```
Введите начальный вклад: 3000
Введите число лет: 10
Введите процентную ставку: 5.5
Через 10 лет вы получите 5124.43 доллара.
```

В конце первого года вы получите 3 000 + (3 000*0.055) = 3165 долларов. В конце следующего года вы получите 3 165 + (3 165*0.055) = 3339.08 долларов. Подобные вычисления удобно производить с помощью цикла for.

8. Напишите программу, которая циклически будет запрашивать ввод пользователем двух денежных сумм, выраженных в фунтах, шиллингах и пенсах (см. упражнения 10 и 12 главы 2). Программа должна складывать введенные суммы и выводить на экран результат, также выраженный в фунтах, шиллингах и пенсах. После каждой итерации программа должна спрашивать пользователя, желает ли он продолжать работу программы. При этом рекомендуется использовать цикл do. Естественной формой взаимодействия программы с пользователем была бы следующая:

```
Введите первую сумму  £5 10 6
Введите первую сумму  £3 2 6
Всего  £8 13 0
Продолжить(y/n)?
```

Для того чтобы сложить две суммы, вам необходимо учесть заем одного шиллинга в том случае, если число пенсов окажется больше 11, и одного фунта, если число шиллингов окажется больше 19.

9. Представьте, что вы собираетесь пригласить к себе шестерых гостей, но за вашим столом могут разместиться всего лишь 4 человека Сколькими способами можно разместить четырех из шести гостей за обеденным столом? Каждый из шести гостей может разместиться на первом стуле. Каждый из оставшихся пяти гостей может занять второй стул. На третьем стуле может разместиться один их четырех гостей, и на четвертом — один из трех оставшихся гостей. Двоим из гостей не достанется ни одного места. Таким образом, число возможных рассадок гостей за столом равно 6*5*4*3 = 360. Напишите программу, которая будет производить аналогичные вычисления для любого числа гостей и любого числа мест за столом (при этом предполагается, что число гостей не меньше числа мест). Программа не должна быть сложной, и вычисление можно организовать с помощью простого цикла for.

10. Модифицируйте программу, описанную в упражнении 7, так, чтобы вместо вычисления текущей суммы на вашем счете она вычисляла, сколько лет потребуется для того, чтобы при заданной процентной ставке и величине начального вклада сумма на вашем счете достигла запрашиваемого вами значения. Для хранения найденного числа лет используйте переменную целого типа (можно отбросить дробную часть значения, полученного в результате расчета). Самостоятельно выберите тип цикла, подходящий для решения задачи.

11. Создайте калькулятор, выполняющий действия над денежными суммами, выраженными в фунтах, шиллингах и пенсах (см. упражнения 10 и 12 главы 2). Калькулятор должен складывать и вычитать вводимые значения, а также производить умножение денежной суммы на вещественное число (операция умножения двух денежных сумм не имеет смысла, по-

скольку квадратных денежных единиц не существует. Деление одной денежной суммы на другую мы тоже не будем рассматривать). Организация взаимодействия с калькулятором описана в упражнении 4 этой главы.

12. Создайте калькулятор, выполняющий четыре арифметических действия над дробями (см. упражнение 9 главы 2 и упражнение 4 этой главы). Формулы, демонстрирующие выполнение арифметических операций над дробями, приведены ниже.

    Сложение: a/b + c/d = (a*d + b*c)/(b*d)
    Вычитание: a/b − c/d = (a*d − b*c)/(b*d)
    Умножение: a/b*c/d = (a*c)/(b*d)
    Деление: a/b/c/d = (a*d)/(b*c)

    Пользователь должен сначала ввести первый операнд, затем знак операции и второй операнд. После вычисления результата программа должна отобразить его на экране и запросить пользователя о его желании произвести еще одну операцию.

Глава 4
Структуры

- Структуры
- Перечисления

В предыдущих главах мы рассматривали только переменные стандартных типов, таких, как float, char и int. Подобные переменные способны представлять какую-либо одну величину — высоту, сумму, значение счетчика и т. д. Однако иногда отдельные переменные бывает удобно объединять в более сложные конструкции. В реальной жизни мы совершаем подобные действия, когда организуем работников фирмы в отделы или составляем предложения из отдельных слов. В C++ одной из конструкций, реализующих объединение разнородных данных, является *структура*.

Структурам посвящена первая часть этой главы. Во второй части мы рассмотрим *перечисления* — другое средство языка C++, определенным образом связанное со структурами.

Структуры

Структура является объединением простых переменных. Эти переменные могут иметь различные типы: int, float и т. д. (как мы позже увидим, именно разнородностью типов переменных структуры отличаются от массивов, в которых все переменные должны иметь одинаковый тип). Переменные, входящие в состав структуры, называются *полями* структуры.

В книгах, посвященных программированию на C, структуры зачастую рассматриваются как одно из дополнительных средств языка и изучаются в конце курса. Для тех, кто изучает язык C++, структуры являются одной из составляющих главных концепций языка — объектов и классов. На самом деле синтаксис структуры фактически идентичен синтаксису класса. На практике отличие структуры от класса заключается в следующем: структуры, как правило, используют в качестве объединения данных, а классы — в качестве объединения данных и функций. Таким образом, изучая структуры, мы тем самым закладываем основы для понимания классов и объектов. Структуры в C++ имеют предназначение, сходное с элементом *запись* языка Pascal и некоторых других языков программирования.

Простая структура

Мы начнем рассмотрение со структуры, содержащей три поля, два из которых имеют целый тип и одно поле — вещественный тип. Эта структура предназначена для хранения информации о комплектующих деталях изделий, выпускаемых фабрикой. Компания производит несколько типов изделий, поэтому номер модели изделия включен в структуру как первое из полей. Номер самой детали представлен вторым полем, а ее стоимость — последним, третьим полем.

В программе PARTS определяется структура part и переменная part1 типа part. Затем полям переменной part присваиваются значения и выводятся на экран.

```
// parts.cpp
// структура для хранения информации о деталях изделий
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
struct part                   // объявление структуры
{
  int modelnumber;            // номер модели изделия
  int partnumber;             // номер детали
  float cost;                 // стоимость детали
};
///////////////////////////////////////////////////////////
int main()
{
  part part1;                 // определение структурной переменной
  part1.modelnumber = 6244;   // инициализация полей
  part1.partnumber = 373;     // переменной part1
  part1.cost = 217.55F;
  // вывод значений полей на экран
  cout << "Модель " << part1.modelnumber;
  cout << ", деталь " << part1.partnumber;
  cout << ", стоимость $" << part1.cost << endl;
  return 0;
}
```

Результат работы программы выглядит следующим образом:

```
Модель 6244, деталь 373, цена $217.55
```

В программе PARTS присутствуют три основных аспекта работы со структурами: определение структуры, определение переменной типа этой структуры и доступ к полям структуры. Давайте подробно рассмотрим каждый из этих аспектов.

Определение структуры

Определение структуры задает ее внутреннюю организацию, описывая поля, входящие в состав структуры:

```
struct part
{
  int modelnumber;
  int partnumber;
  float cost;
};
```

Синтаксис определения структуры

Определение структуры начинается с ключевого слова struct. Затем следует имя структуры, в данном случае этим именем является part. Объявления полей структуры modelnumber, partnumber и cost заключены в фигурные скобки. После закрывающей фигурной скобки следует точка с запятой (;) — символ, означающий конец определения структуры. Обратите внимание на то, что использование точки с запятой после блока операторов при определении структуры отличает синтаксис структуры от синтаксиса других рассмотренных нами элементов программы. Как мы видели, в циклах, ветвлениях и функциях блоки операторов тоже ограничивались фигурными скобками, однако точка с запятой после таких блоков не ставилась. Синтаксис структуры представлен на рис. 4.1.

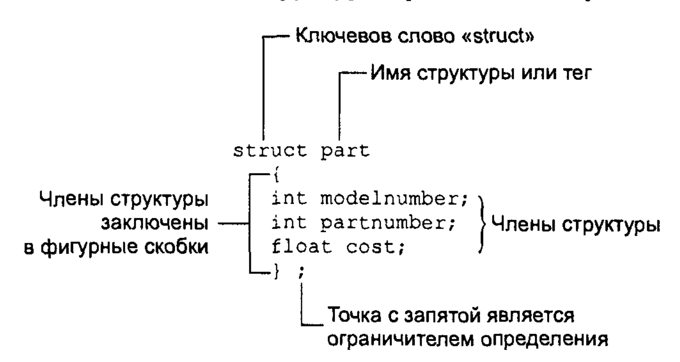


Рис. 4.1. Синтаксис определения структуры

Смысл определения структуры

Определение структуры part необходимо для того, чтобы создавать на его основе переменные типа part. Само определение не создает никаких переменных; другими словами, не происходит ни выделения физической памяти, ни объявления переменной. В то же время определение обычной переменной предполагает выделение памяти под эту переменную. Таким образом, определение структуры фактически задает внутреннюю организацию структурных переменных после того, как они будут определены. Рисунок 4.2 иллюстрирует вышесказанное.

Не удивляйтесь, если связь между структурой и структурными переменными покажется вам сходной со связью между классами и объектами, о которой шла речь в главе 1 «Общие сведения». Как мы позже увидим, объект действительно имеет такое же отношение к своему классу, как структурная переменная к своей структуре.

Определение структурной переменной

Первый оператор функции main() выглядит следующим образом:

```
part part1;
```

Он представляет собой определение переменной part1, имеющей тип part. Определение переменной означает, что под эту переменную выделяется память.

Сколько же памяти выделяется в данном случае? Под структурную переменную всегда отводится столько памяти, сколько достаточно для хранения всех ее полей. В нашем примере необходимо выделить по 4 байта для двух полей типа int (если считать, что операционная система является 32-битной) и 4 байта для поля типа float. На рис. 4.3 изображено расположение в памяти переменной part1.

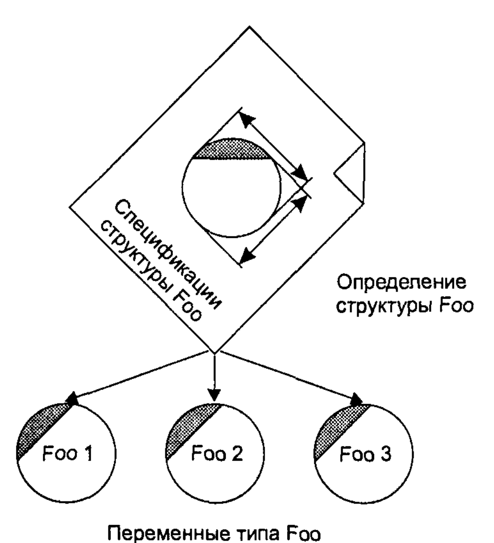


Рис. 4.2. Структуры и их переменные

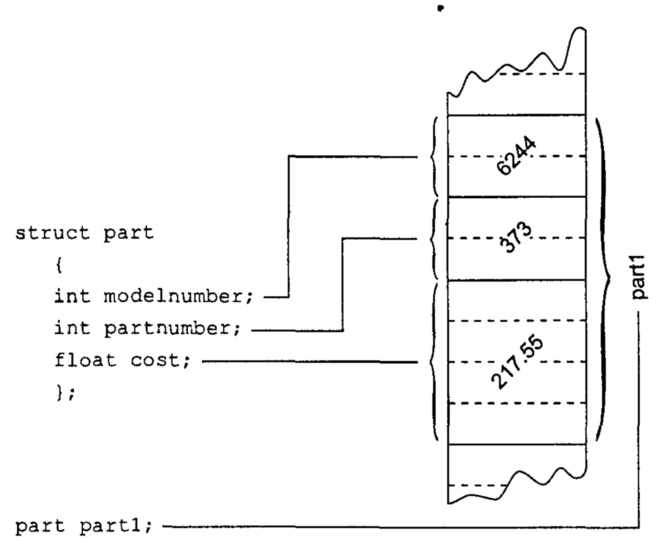


Рис. 4.3. Расположение членов структуры в памяти

В некотором смысле мы можем считать структуру part новым типом данных. Этот вопрос станет более понятен при дальнейшем рассмотрении, однако уже

сейчас вы можете заметить, что определение структурной переменной по своему синтаксису идентично определению переменной стандартного типа:

```
part part1;
int var1;
```

Такое сходство не является случайным. Одна из целей языка C++ — сделать работу с пользовательскими типами данных максимально похожей на работу со стандартными типами данных (в языке С при определении структурных переменных необходимо применять ключевое слово struct; определение переменной part1 на языке С выглядело бы как struct part part1).

Доступ к полям структуры

Когда структурная переменная определена, доступ к ее полям возможен с применением пока неизвестной нам *операции точки*. В программе первому из полей структуры присваивается значение при помощи оператора

```
part1.modelnumber = 6244;
```

Поле структуры идентифицируется с помощью трех составляющих: имени структурной переменной part1, операции точки (.) и имени поля modelnumber. Подобную запись следует понимать как «поле modelnumber переменной part1». Операция точки в соответствии с общепринятой терминологией называется операцией *доступа к полю структуры*, но, как правило, такое длинное название не употребляют.

Обратите внимание на то, что в выражении с использованием операции точки (.) на первом месте стоит не название структуры (part), а имя структурной переменной (part1). Имя переменной нужно для того, чтобы отличать одну переменную от другой: part1 от part2 и т. д., как показано на рис. 4.4.

С полями структурной переменной можно обращаться так же, как с обычными простыми переменными. Так, в результате выполнения оператора

```
part1.modelnumber = 6244;
```

полю modelnumber присваивается значение 6244 при помощи обычной операции присваивания. В программе также продемонстрирован вывод значения поля на экран с помощью cout:

```
cout << "\nМодель " << part1.modelnumber;
```

Другие возможности структур

Структуры обладают достаточно широким набором возможностей. Рассмотрим некоторые из них.

Инициализация полей структуры

Следующий пример демонстрирует способ, при помощи которого можно инициализировать поля предварительно определенной структурной переменной. В программе используются две структурные переменные.

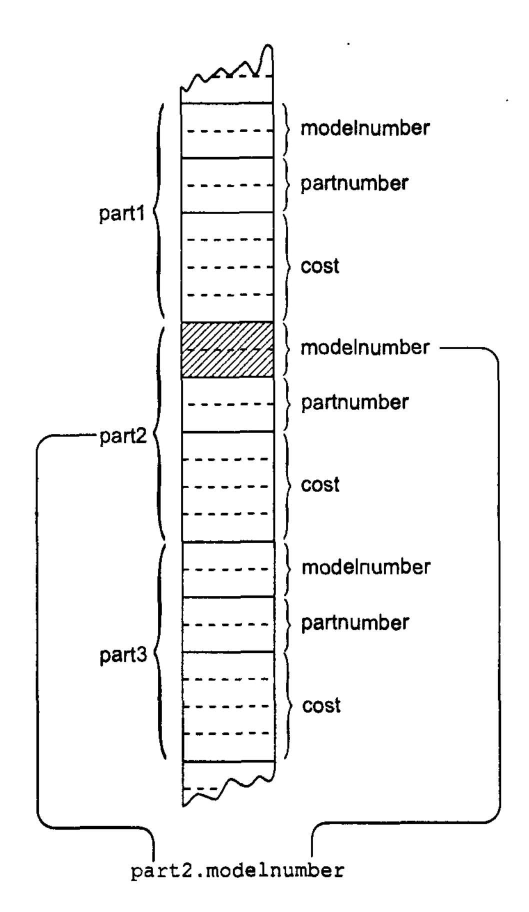


Рис. 4.4. Операция «точка»

```
// partinit.cpp
// инициализация структурных переменных
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
struct part             // объявление структуры
{
  int modelnumber;      // номер модели изделия
  int partnumber;       // номер детали
  float cost;           // стоимость детали
};
///////////////////////////////////////////////////////////
int main()
{                       // инициализация переменной
  part part1 = {6244,373,217.55F };
  part part2;           // объявление переменной
                        // вывод полей первой переменной
  cout << "Модель " << part1.modelnumber;
  cout << ", деталь " << part1.partnumber;
  cout << ", стоимость $" << part1.cost << endl;
  part2 = part1;        // присваивание структурных переменных
```

```
  // вывод полей второй переменной
  cout << "Модель " << part2.modelnumber;
  cout << ", деталь " << part2.partnumber;
  cout << ", стоимость $" << part2.cost <<endl;
  return 0;
}
```

В приведенной программе определены две переменные типа part: part1 и part2. После того как поля переменной part1 инициализированы, происходит вывод их значений на экран, затем значение переменной part1 присваивается переменной part2, и значения ее полей также выводятся на экран. Результат работы программы выглядит следующим образом:

```
Модель 6244, часть 373, цена $217.55
Модель 6244, часть 373, цена $217.55
```

Неудивительно, что обе выведенные строки идентичны, поскольку значение второй переменной было присвоено первой переменной.

Инициализация полей переменной part1 производится в момент ее определения:

```
part part1 = { 6244, 373, 217.55 };
```

Значения, которые присваиваются полям структурной переменной, заключены в фигурные скобки и разделены запятыми. Первая из величин присваивается первому полю, вторая — второму полю и т. д.

Присваивание структурных переменных

Как мы видим из программы PARTINT, мы можем присваивать значение одной структурной переменной другой структурной переменной:

```
part2 = part1;
```

Значение каждого поля переменной part1 присваивается соответствующему полю переменной part2. Поскольку в больших структурах число полей иногда может измеряться десятками, для выполнения присваивания структурных переменных компилятору может потребоваться проделать большое количество работы.

Обратите внимание на то, что операция присваивания может быть выполнена только над переменными, имеющими один и тот же тип. В случае попытки выполнить операцию присваивания над переменными разных типов компилятор выдаст сообщение об ошибке.

Пример применения структур

Рассмотрим теперь, каким образом можно применить способность структуры объединять данные различных типов. Возможно, вы знаете, что в английской системе мер основными единицами измерения длины служат фут и дюйм, причем один фут равен 12 дюймам. Расстояние, равное 15 футам и 8 дюймам, записывается как 15'-8". Дефис в данной записи не означает знак «минус» , а служит для разделения значений футов и дюймов. Иллюстрация к английской системе мер приведена на рис. 4.5.

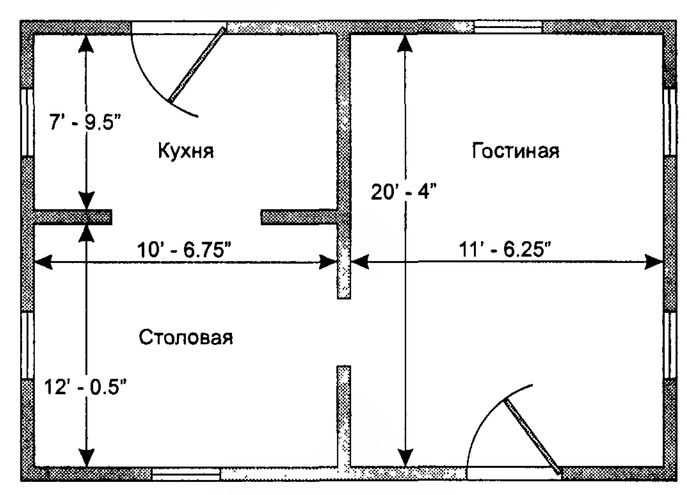


Рис. 4.5. Английская система мер

Предположим, вам необходимо создать инженерный проект с использованием английской системы мер. Было бы удобно представлять длины в виде двух чисел, одно из которых равнялось бы количеству футов, а другое — количеству дюймов. В следующем примере, ENGLSTRC, такая идея реализована с помощью структуры. Мы покажем, как можно при помощи типа Distance складывать две длины.

```
// englstrc.cpp
// английская система мер, реализованная с помощью структуры
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
struct Distance              // длина в английской системе
{
  int feet;
  float inches;
};
//////////////////////////////////////////////////////////
int main()
{
  Distance d1, d3;           // определения двух переменных
  Distance d2 = { 11, 6.25 }; // определение с инициализацией
  // ввод полей переменной d1
  cout << "\nВведите число футов: "; cin >> d1.feet;
  cout << "Введите число дюймов: "; cin >> d1.inches;
  // сложение длин d1 и d2 и присвоение результата d3
  d3.inches = d1.inches + d2.inches; // сложение дюймов
  d3.feet = 0;                       // с возможным заемом
  if( d3.inches >= 12.0 )            // если сумма больше 12.0,
  {                                  // то уменьшаем
  d3.inches -= 12.0;                 // число дюймов на 12.0 и
  d3.feet++;                         // увеличиваем число футов на 1
  }
```

```
    d3.feet += d1.feet + d2.feet;       // сложение футов
    // вывод всех длин на экран
    cout << d1.feet <<"\'-" << d1.inches << "\" + ";
    cout << d2.feet <<"\'-" << d2.inches << "\" = ";
    cout << d3.feet <<"\'-" << d3.inches << "\"\n";
    return 0;
}
```

Здесь структура Distance состоит из двух полей: feet и inches. Число дюймов не обязательно является целым, поэтому для его хранения мы используем тип float. Число футов, наоборот, всегда является целым, и поэтому его можно хранить в переменной типа int.

Мы определяем переменные типа Distance, d1, d2 и d3, причем переменную d2 мы инициализируем значением 11'-6.25", а переменные d1 и d3 пока оставлены без начального значения. Далее программа запрашивает у пользователя число футов и дюймов, которые используются для инициализации переменной d1 (при этом число дюймов должно быть меньше 12.0). Затем значения d1 и d2 складываются, а результат присваивается переменной d3. Наконец, программа выводит две первоначальные длины и их сумму. Примером результата работы программы может служить следующий:

```
Введите число футов: 10
Введите число дюймов: 6.75
10'-6.75" + 11'-6.25" = 22'-1"
```

Обратите внимание на то, что мы не можем сложить две длины следующим образом:

```
d3 = d1 + d2;                    // некорректное действие
```

Такое действие вызовет ошибку потому, что в C++ не определены действия, которые необходимо выполнить при сложении переменных типа Distance. Операция + применима для стандартных типов C++, но она не определена для типов данных, которые создаются пользователями (как мы увидим в главе 8 «Перегрузка операций», одним из достоинств классов является возможность определять сложение и другие операции над пользовательскими типами данных).

Вложенные структуры

Структуры допускают вложенность, то есть использование структурной переменной в качестве поля какой-либо другой структуры. Внесем дополнения в программу ENGLSTRC, которые позволят это продемонстрировать. Пусть нам необходимо хранить данные о размерах комнаты, то есть ее ширину и длину. Поскольку мы работаем с английской системой мер, нам придется хранить эти величины с использованием типа Distance:

```
struct Room
{
  Distance length;
  Distance width;
}
```

Следующий наш пример, ENGLAREA, использует структуру Room:

```
// englarea.cpp
// использование вложенных структур
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
struct Distance           // длина в английской системе
{
  int feet;
  float inches;
};
//////////////////////////////////////////////////////////
struct Room              // размеры прямоугольной комнаты
{
  Distance length;        // длина
  Distance width;         // ширина
};
//////////////////////////////////////////////////////////
int main()
{
  Room dining;            // переменная
  dining.length.feet = 13; // задание параметров комнаты
  dining.length.inches = 6.5;
  dining.width.feet = 10;
  dining.width.inches = 0.0;
  // преобразование длины и ширины в вещественный формат
  float l = dining.length.feet + dining.length.inches/12;
  float w = dining.width.feet + dining.width.inches/12;
  // вычисление площади комнаты и вывод на экран
  cout << "Площадь комнаты равна " << l * w
  << " квадратных футов\n";
  return 0;
}
```

В данной программе используется единственная переменная dining, имеющая тип Room:

```
Room dining;             // переменная dining типа Room
```

Затем программа производит различные действия с полями этой структурной переменной.

Доступ к полям вложенных структур

Если одна структура вложена в другую, то для доступа к полям внутренней структуры необходимо дважды применить операцию точки (.):

```
dining.length.feet = 13;
```

В этом операторе dining — имя структурной переменной, как и раньше; length — имя поля внешней структуры Room; feet — имя поля внутренней структуры Distance. Таким образом, данный оператор берет поле feet поля length переменной dining и присваивает этому полю значение, равное 13. Иллюстрация этого механизма приведена на рис. 4.6.

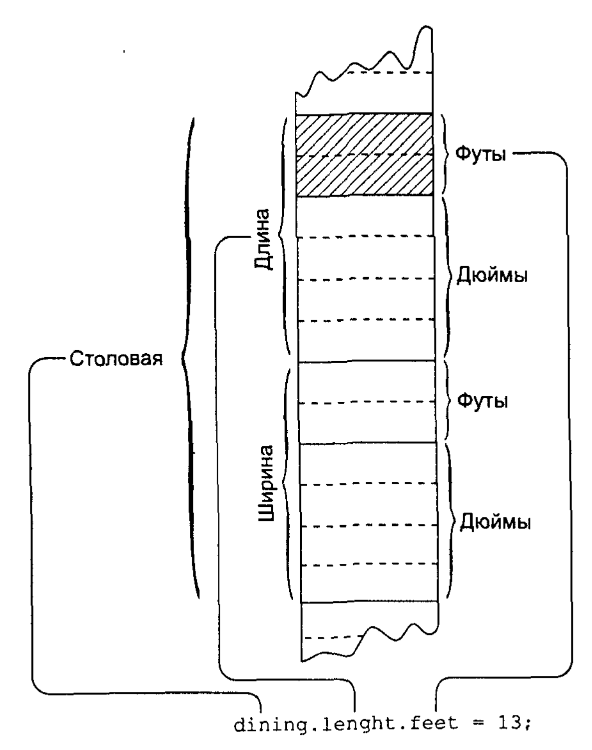


Рис. 4.6. Операция «точка» и вложенные структуры

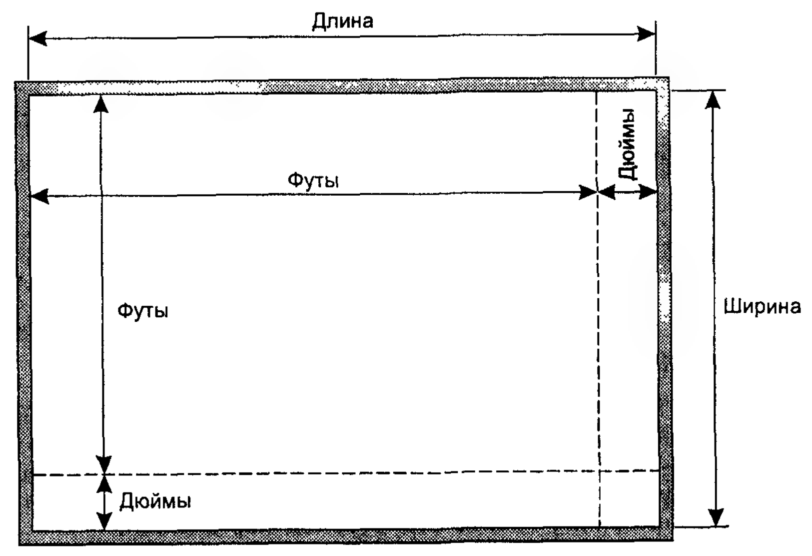


Рис. 4.7. Размеры помещения в футах и дюймах

Когда поля переменной dining инициализированы, программа подсчитыв площадь комнаты, как показано на рис. 4.7.

Чтобы найти площадь комнаты, программе необходимо выразить значения длины и ширины в футах. Для этого используются переменные l и w типа float. Их значения получаются путем сложения числа футов соответствующего измерения с числом дюймов, деленным на 12. Преобразование целого типа поля feet к типу float произойдет автоматически, и результат сложения будет иметь тип float. Затем переменные l и w перемножаются, в результате чего мы получаем значение площади комнаты.

Преобразования пользовательских типов данных

Обратите внимание на то, что программа преобразовывает две величины типа Distance к двум величинам типа float. Кроме того, при вычислении площади комнаты на самом деле осуществляется преобразование переменной типа Room, хранящей два значения типа Distance, к одному вещественному значению. Результат работы программы выглядит так:

```
Площадь комнаты равна 135.416672 квадратных футов
```

Возможность преобразования значений одного типа в значения другого типа является важным аспектом для программ, которые используют типы данных, определяемые пользователем.

Инициализация и вложенные структуры

Каким образом инициализировать структурную переменную, которая содержит внутри себя поле, также являющееся структурной переменной? Следующий оператор инициализирует переменную dining теми же значениями, что и в последней программе:

```
Room dining = { {13, 6.5}, {10, 0.0} };
```

Каждая структура Distance, входящая в состав типа Room, инициализируется отдельно. Как мы уже говорили, значения, которыми инициализируется структурная переменная, заключаются в фигурные скобки и разделяются запятыми. Первая из внутренних структур Distance инициализируется с помощью конструкции

```
{13, 6.5}
```

а вторая — с помощью конструкции

```
{10, 0.0}
```

Инициализировав по отдельности каждое из полей типа Distance, мы инициализируем переменную типа Room, заключив значения типа Distance в фигурные скобки и разделив их запятыми.

Глубина вложения

Теоретически не существует ограничения на величину уровня вложенности структур. В программе, работающей с дизайном квартир, вы встретите такое обращение к полю структурной переменной:

```
apartment1.laundry_room.washing_machine.width.feet
```

Пример карточной игры

Давайте рассмотрим еще один пример. В нем с помощью структуры будет создана модель игральной карты. Эта программа имитирует действия уличного игрока. Игрок показывает вам три карты, перемешивает их и раскладывает. Если вам удастся угадать, в какой последовательности разложены карты, то вы выиграли. Все происходит у вас на виду, но игрок мешает карты настолько быстро и умело, что вы путаетесь в расположении карт и, как правило, проигрываете свою ставку.

Структура, хранящая информацию об игральной карте, выглядит так:

```
struct card
{
  int number;
  int suit;
};
```

Эта структура содержит два поля, number и suit, предназначенных для хранения соответственно достоинства карты и ее масти. Достоинство карты определяется числом от 2 до 14, где числа 11, 12, 13 и 14 означают соответственно валета, даму, короля и туза. Масть карты — число от 0 до 3, где 0 означает трефовую масть, 1 — бубновую, 2 — червовую, 3 — пиковую.

Листинг программы CARDS выглядит следующим образом[1]:

```
// cards.cpp
// представление игральных карт при помощи структуры
#include <iostream>
using namespace std;
const int clubs = 0;          // масти
const int diamonds = 1;
const int hearts = 2;
const int spades = 3;
const int jack = 11;          // достоинства карт
const int queen = 12;
const int king = 13;
const int ace = 14;
//////////////////////////////////////////////////////////
struct card
{
  int number;                 // достоинство
  int suit;                   // масть
};
//////////////////////////////////////////////////////////
int main()
{
  card temp,chosen,prize;     // три карты
  int position;
  card card1 = { 7, clubs };  // инициализация карты 1
  cout << "Карта 1: 7 треф\n";
  card card2 = { jack, hearts }; // инициализация карты 2
  cout << "карта 2: валет червей\n";
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
  card card3 = { ace, spades };   // инициализация карты 3
  cout << "Карта 3: туз пик\n ";
  prize = card3;                  // запоминание карты 3
  cout << "Меняем местами карту 1 и карту 3...\n";
  temp = card3; card3 = card1; card1 = temp;
  cout << "Меняем местами карту 2 и карту 3...\n";
  temp = card3; card3 = card2; card2 = temp;
  cout << "Меняем местами карту 1 и карту 2...\n";
  temp = card2; card2 = card1; card1 = temp;
  cout << "На какой позиции (1,2 или 3) теперь туз пик? ";
  cin >> position;
  switch (position)
  {
    case 1: chosen = card1; break;
    case 2: chosen = card2; break;
    case 3: chosen = card3; break;
  }
  if(chosen.number == prize.number && // сравнение карт
  chosen.suit == prize.suit)
    cout << "Правильно! Вы выиграли!\n";
  else
    cout << "Неверно. Вы проиграли.\n";
  return 0;
}
```

Вот пример возможного взаимодействия с программой:

```
// вывод программы
Карта 1: 7 треф
Карта 2: валет червей
Карта 3: туз пик
Меняем местами карту 1 и карту 3
Меняем местами карту 2 и карту 3
Меняем местами карту 1 и карту 2
На какой позиции (1,2 или 3) теперь туз пик? 3
Неверно. Вы проиграли.
```

В данном случае незадачливый игрок выбрал неверную карту (правильным ответом было число 2).

Программа начинается с объявления нескольких переменных типа const int, предназначенных для хранения величин, соответствующих четырем мастям и четырем достоинствам, имеющим словесное (нецифровое) название. В данной программе используются не все из объявленных переменных; «лишние» объявления сделаны в целях сохранения общности алгоритма. Далее в программе определяется структура card и три неинициализируемые структурные переменные temp, chosen и prize. Позже определяются и инициализируются еще три структурные переменные: card1, card2 и card3, играющие роль трех карт, предназначенных для угадывания. Информация об этих картах выводится игроку на экран. Далее одна из карт запоминается с помощью переменной prize, и эту карту будет предложено угадать игроку в конце игры.

Теперь программа перемешивает карты. Она меняет местами первую и третью карты, затем вторую и третью карты, и наконец, первую и вторую карты. При этом программа сообщает игроку обо всех перестановках (если такой вариант

игры покажется вам слишком простым, можете добавить в программу еще несколько операций по перемешиванию карт. Кроме того, можно усложнить правила игры, ограничив время, в течение которого сообщение о перестановке карт будет присутствовать на экране).

Программа спрашивает пользователя, где находится одна из карт. Информация о карте, находящейся в выбранной позиции, записывается в переменную chosen, значение которой сравнивается со значением переменной prize. Если эти значения совпадают, то игрок выигрывает; в противном случае игрок считается проигравшим.

Обратите внимание на то, насколько простой оказывается реализация перемешивания карт:

```
temp = card3;   card3 = card1;   card1 = temp;
```

Благодаря возможности выполнения операции присваивания над структурными переменными процесс перемешивания карт в программе очень похож на аналогичный процесс в реальной жизни.

К сожалению, сравнивать между собой структурные переменные, как и складывать их друг с другом, нельзя. Конструкция

```
if ( chosen == prize )   // некорректно
```

недопустима потому, что операции == ничего не известно о переменных типа card. Однако, как мы увидим позже, данная проблема разрешима с помощью перегрузки операций.

Структуры и классы

Мы должны признать, что утаили от вас некоторые сведения, касающиеся возможностей структур. Мы были правы, когда говорили, что структуры обычно используются для объединения данных, а классы — для объединения данных и функций. Тем не менее, в C++ структуры также обладают возможностью включать в себя функции (что, правда, неверно в отношении языка C). Различия в синтаксисе между структурами и классами минимальны, поэтому теоретически можно использовать одно вместо другого. Однако структуры обычно используют только для хранения данных, а классы — для хранения данных и функций. Мы разберем этот вопрос подробнее в главе 6 «Объекты и классы».

Перечисления

Как мы уже видели, структуры можно рассматривать в качестве нового типа данных, определяемого пользователем. Кроме структур, существует еще один способ создания пользовательских типов данных, называемый *перечислением*. Этот способ имеет гораздо меньшую практическую важность, чем структуры, и для того, чтобы писать хорошие объектно-ориентированные программы на языке C++, нет необходимости применять перечисления. И все же перечисления отно-

сятся к типичным средствам программирования в стиле C++, и их использование может заметно упростить процесс создания вашей программы.

Дни недели

Перечисления используются в тех случаях, когда переменные создаваемого типа могут принимать заранее известное конечное (и, как правило, небольшое) множество значений. Приведем простой пример, DAYENUM, в котором перечисления используются для создания типа данных, хранящего дни недели.

```
// dayenum.cpp
// применение перечислений
#include <iostream>
using namespace std;
// объявление перечисляемого типа
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
int main()
{
  days_of_week day1, day2;      // определения переменных,
                                // хранящих дни недели
  day1 = Mon;                   // инициализация переменных
  day2 = Thu;
  int diff = day2 - day1;       // арифметическая операция
  cout << "Разница в днях: " << diff << endl;
  if( day1 < day2)              // сравнение
  cout << "day1 наступит раньше, чем day2\n";
  return 0;
}
```

Объявление типа начинается со слова enum и содержит перечисление всех возможных значений переменных создаваемого типа. Эти значения называются *константами перечисляемого типа*. Перечисляемый тип days_of_week включает 7 констант перечисляемого типа: Sun, Mon, Tue и т. д. до Sat. На рис. 4.8 приведен синтаксис объявления перечисляемого типа.

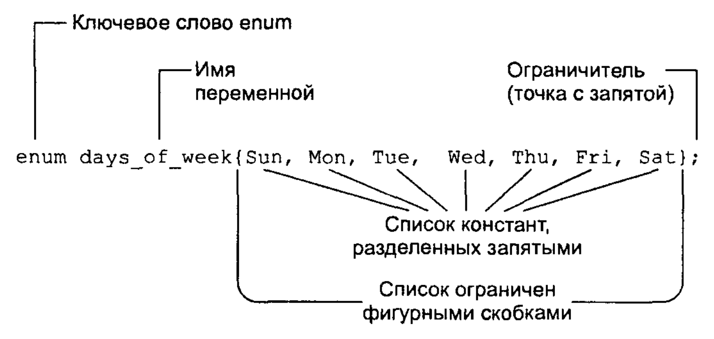


Рис. 4.8. Синтаксис спецификатора enum

Итак, перечисление представляет собой список всех возможных значений. В этом отношении тип int отличается от перечислений, поскольку он задается

с помощью диапазона значений. Используя перечисления, необходимо давать имя каждому возможному значению создаваемого типа. Различие между типом int и перечисляемыми типами проиллюстрировано на рис. 4.9.

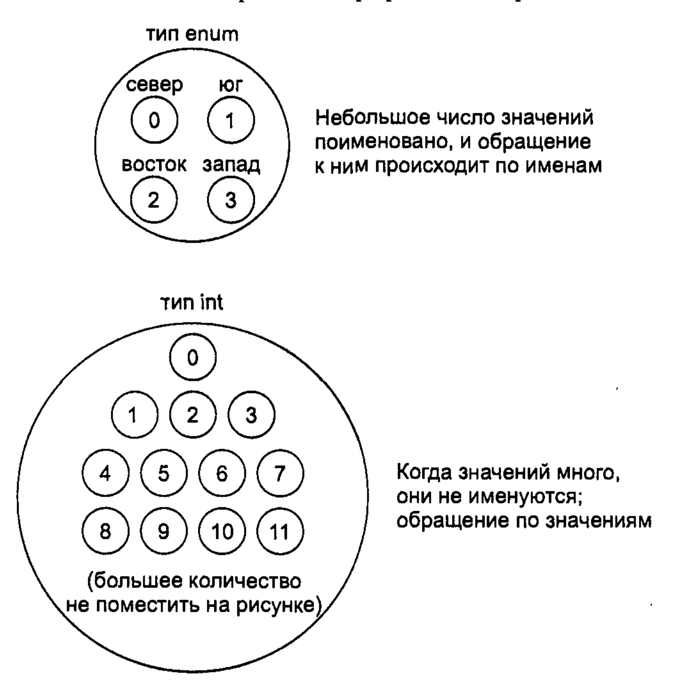


Рис. 4.9. Использование типов int и enum

Когда тип days_of_week определен, можно определять и переменные этого типа. В программе DAYENUM мы используем две переменные типа days_of_week:

```
days_of_week dayl, day2;
```

В языке С в подобных объявлениях было необходимо использовать ключевое слово enum:

```
enum days_of_week dayl, day2;
```

но язык С++ устранил эту необходимость.

Переменным перечисляемого типа можно присваивать любое из значений, указанных при объявлении типа. Например, переменным dayl и day2 из нашей последней программы мы присваиваем значения Mon и Thu. Присваивание значений, не указанных при перечислении, например

```
dayl = halloween;
```

не допускается.

Перечисляемые типы данных допускают применение основных арифметических операций. В частности, в нашей программе производится деление двух переменных перечисляемого типа. Кроме того, перечисляемые типы допускают

применение операции сравнения, что также используется в нашем примере. Результат работы программы DAYENUM выглядит следующим образом:

```
Разница в днях: 3
day1 наступит раньше, чем day2
```

Использование арифметических операций и операций отношения с перечисляемыми типами данных, как правило, не несет большой смысловой нагрузки. Например, если мы определили тип данных pets, хранящий названия домашних животных, следующим образом:

```
enum pets { cat, dog, hamster, canary, ocelot };
```

то неясен смысл выражений dog + canary или (cat < hamster) и т. п.

Внутренне перечисляемые типы данных представляются и обрабатываются как целые числа. Этот факт объясняет возможность применения к переменным перечисляемого типа арифметических операций и операций отношения. Фактически первое значение в списке рассматривается как число 0, второе — как число 1, и т. д. В программе DAYENUM значения от Sun до Sat представляют собой целые числа в диапазоне 0...6.

Арифметические операции над переменными перечисляемых типов сводятся к операциям над целыми числами. Однако, несмотря на то, что компилятор знает о целочисленной форме представления перечисляемых значений, стоит использовать этот факт с большой осторожностью. Если вы попробуете выполнить присваивание

```
day1 = 5;
```

компилятор выдаст предупреждающее сообщение (хотя компиляция произойдет без ошибок). Рекомендуется там, где возможно, не использовать целочисленную интерпретацию перечисляемых значений.

Перечисления и программа подсчета числа слов

Наша следующая программа подсчитывает количество слов в предложении, вводимом с клавиатуры пользователем. Структура работы программы показана на рис. 4.10.

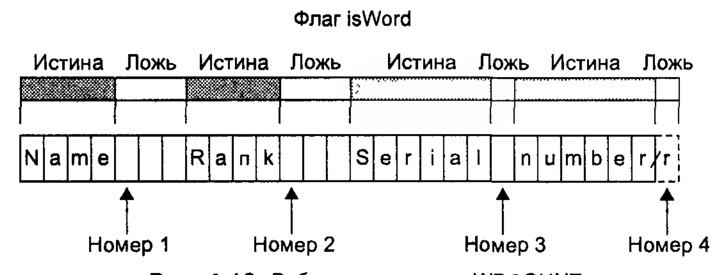


Рис. 4.10. Работа программы WDCOUNT

В отличие от примера CHCOUNT, здесь мы уже не будем сводить подсчет слов к подсчету введенных пробелов (хотя наша программа все еще не будет обраба-

тывать табуляции и прочие разделяющие символы). Мы используем перечисляемый тип данных с двумя значениями. Вот листинг программы WDCOUNT:

```
// wdcount.cpp
// подсчет числа слов в предложении с использованием перечислений
#include <iostream>
using namespace std;
#include <conio.h>              // для getche()
enum itsaWord { NO, YES };      // NO=0, YES=1
int main()
{
  itsaWord isWord = NO;         // isWord равно YES, когда
  // вводится слово, и NO, когда вводятся пробелы
  char ch='a';                  // считывание символа с клавиатуры
  int wordcount = 0;            // число слов
  cout << "Введите предложение: \n";
  do {
    ch = getche();              // ввод символа
    if( ch==' ' || ch=='\r' )  // если введен пробел,
    {
      if( isWord == YES )       // а до этого вводилось слово,
      {                         // значит, слово закончилось
        wordcount++;            // учет слова
        isWord = NO;            // сброс флага
      }
    }                           // в противном случае
    else                        // ввод слова продолжается
    if ( isWord == NO )         // если начался ввод слова,
    isWord = YES;               // то устанавливаем флаг
    } while( ch !='\r ' );      // выход по нажатию Enter
  cout << "\n---Число слов: " << wordcount << "---\n";
  return 0;
}
```

Программа считывает символы с клавиатуры в цикле do. Она не выполняет никаких дополнительных действий над непробельными символами. Когда с клавиатуры вводится пробел, счетчик количества слов увеличивается на единицу, после чего программа игнорирует все пробелы, вводимые до появления первого непробельного символа. Далее программа снова «пропускает» непробельные символы до появления пробела, увеличивает счетчик количества слов на единицу и т. д. Для того чтобы реализовать подобный алгоритм, программе необходимо различать, вводится ли в данный момент слово или последовательность пробелов. Для этого используется переменная isWord перечисляемого типа itsaWord, определенного следующим образом:

```
enum itsaWord { NO, YES };
```

Переменные типа itsaWord могут иметь только два различных значения: NO и YES. Обратите внимание на то, что перечисление начинается со значения NO; это означает, что этому значению будет соответствовать внутреннее представление в виде целого числа 0, которое, как мы знаем, может интерпретироваться еще и как ложное значение (мы могли бы использовать вместо перечисляемого типа стандартный тип bool).

В начале работы программы переменной isWord присваивается значение NO. Как только с клавиатуры буден получен первый непробельный символ, значе-

ние переменной изменится на YES и будет сигнализировать о том, что в данный момент вводится слово. Значение YES будет сохраняться за переменной isWord до тех пор, пока с клавиатуры не будет введен пробел. Ввод пробела означает конец слова, поэтому переменная isWord получит значение NO, которое будет сохраняться до тех пор, пока не появится непробельный символ. Мы знаем, что значения NO и YES на самом деле представлены как 0 и 1, однако не используем этот факт. Мы могли бы использовать запись if(isWord) вместо if(isword == YES) и if(!isWord) вместо if(isword == NO), но это не является хорошим стилем программирования.

Обратите внимание на то, что нам необходимы дополнительные фигурные скобки вокруг второго из операторов if, чтобы последующий else сочетался не со вторым if, а с первым.

Для того чтобы организовать флаг в ситуациях, подобных рассмотренной в программе WDCOUNT, можно воспользоваться переменной типа bool. В зависимости от контекста задачи такой способ может оказаться проще, чем использование перечислений.

Пример карточной игры

Рассмотрим еще один, последний, пример использования перечисляемых типов данных. Вспомним, что в программе CARDS, созданной нами, использовалась группа констант типа const int, предназначенных для представления карточных мастей:

```
const int clubs = 0;
const int diamonds = 1;
const int hearts = 2;
const int spades = 3;
```

Такой способ задания констант, как можно видеть, не отличается изящностью. Давайте перепишем код программы CARDS, используя перечисления.

```
// cardenum.cpp
// карточная игра с использованием перечислений
#include <iostream>
using namespace std;
const int jack = 11;           // именованные достоинства карт
const int queen = 12;
const int king = 13;
const int ace = 14;
enum Suit { clubs, diamonds, hearts, spades };
//////////////////////////////////////////////////////////
struct card
{
  int number;                  // достоинство карты
  Suit suit;                   // масть
};
//////////////////////////////////////////////////////////
int main()
{
  card temp, chosen, prize;    // определение карт
  int position;
  card card1 = { 7, clubs };   // инициализация card1
  cout << "Карта 1: 7 треф\n";
  card card2 = { jack, hearts }; // инициализация card2
```

```
  cout << "Карта 2: валет червей\n";
  card card3 = { ace, spades };   // инициализация card3
  cout << "Карта 3: туз пик\n";
  prize = card3;                  // запоминаем карту 3
  cout << "Меняем местами карту 1 и карту 3\n";
  temp = card3; card3 = card1; card1 = temp;
  cout << "Меняем местами карту 2 и карту 3\n";
  temp = card3; card3 = card2; card2 = temp;
  cout << "Меняем местами карту 1 и карту 2\n";
  temp = card2; card2 = card1; card1 = temp;
  cout << "На какой позиции (1,2 или 3) теперь туз пик?";
  cin >> position;
  switch (position)
  {
    case 1: chosen = card1; break;
    case 2: chosen = card2; break;
    case 3: chosen = card3; break;
  }
  if( chosen.number == prize.number && // сравнение карт
  chosen.suit == prize.suit)
    cout << "Правильно! Вы выиграли!\n";
  else
    cout << "Неверно. Вы проиграли.\n ";
  return 0;
}
```

Мы заменили серию констант, обозначающих масти, на перечисляемый тип:

```
enum Suit { clubs, diamonds, hearts, spades };
```

Такой подход упрощает использование констант, поскольку нам известны все возможные значения мастей, и попытка использования некорректного значения, например

```
card1.suit = 5;
```

вызовет предупреждающее сообщение от компилятора.

Задание целых значений для перечисляемых констант

Как мы говорили, первой из перечисляемых констант соответствует целое значение, равное 0, второй — значение, равное 1, и т. д. Для того чтобы изменить значение, с которого начинается нумерация, можно с помощью операции присваивания задать это значение первой из перечисляемых констант:

```
enum Suit { clubs = 1, diamonds, hearts, spades };
```

В этом случае следующим по списку константам будут соответствовать числа 2, 3 и 4 соответственно. Фактически мы можем изменить величину целого числа, соответствующего любому элементу списка, применив к нему операцию присваивания, как это было показано выше.

Недостаток перечислений

Важным недостатком перечисляемых типов данных является то, что они не распознаются средствами ввода/вывода C++. Например, результатом вывода в фрагменте

```
enum direction { north, south, east, west };
direction dir = south;
cout << dir1;
```

будет не south, формально являющееся значением переменной dir, а ее внутреннее представление, то есть целое число 1.

Примеры перечисляемых типов

Приведем несколько примеров перечисляемых типов данных, которые, возможно, помогут вам лучше понять область наилучшего применения этого средства языка C++:

```
enum months { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
                              // месяцы года
enum switch { off, on };      // переключатель
enum meridian { am, pm };     // меридиан
enum chess { pawn, knight, bishop, rook, queen, king };
                              // шахматные фигуры
enum coins { penny, nickel, dime, quarter, half-dollar, dollar };
                              // монеты
```

Мы еще встретимся с применением перечислений в других разделах нашей книги.

Резюме

В этой главе мы познакомились с двумя средствами C++: структурами и перечислениями. Структура является важным аспектом C++, поскольку ее синтаксис совпадает с синтаксисом класса. Классы (по крайней мере, с точки зрения синтаксиса) представляют собой не что иное, как структуры, включающие в себя функции. Типичной целью использования структур является объединение данных различных типов в один программный элемент. В определении структуры содержится информация о том, какие поля находятся внутри нее. При определении структурных переменных выделяется память, необходимая для размещения значений всех полей такой переменной. Содержимое структурных переменных в некоторых ситуациях рассматривается как единое целое, например при присваивании значения одной структурной переменной другой структурной переменной, но в остальных случаях существует возможность доступа к отдельным полям структурной переменной. Как правило, обращение к полю структурной переменной производится с помощью операции точки (.).

Перечисление представляет собой тип данных, определяемый пользователем, значения которого задаются с помощью списка. При объявлении типа указывается его имя и список значений, каждое из которых представлено константой перечисляемого типа. После объявления типа можно определять переменные, имеющие этот тип. Компилятор использует внутреннее представление значений констант перечисляемого типа в виде целых чисел.

Не следует смешивать между собой понятия структуры и перечисления. Структура представляет собой мощное и гибкое средство, позволяющее объединять данные самых разных типов. Перечисления всего лишь позволяют определять переменные, принимающие фиксированный перечисляемый набор значений.

Вопросы

Ответы на приведенные вопросы можно найти в приложении Ж.

1. Структура объединяет:

   а) данные одного типа;

   б) логически связанные данные;

   в) целые именованные значения;

   г) переменные.

2. Истинно ли следующее утверждение: структура и класс имеют схожий синтаксис?

3. После закрывающей фигурной скобки структуры ставится _________.

4. Опишите структуру, содержащую три переменные типа int с названиями hrs, mins и secs. Назовите структуру именем time.

5. Истинно ли следующее утверждение: при определении структуры выделяется память под переменную?

6. При обращении к полю структуры левым операндом операции (.) является:

   а) поле структуры;

   б) имя структуры;

   в) структурная переменная;

   г) ключевое слово struct.

7. Напишите оператор, присваивающий полю hrs структурной переменной time2 значение, равное 11.

8. Сколько байтов памяти займут три структурные переменные типа time, если структура time содержит три поля типа int?

9. Напишите определение, инициализирующее поля структурной переменной time1 типа time, описанной в вопросе 4, значениями hrs=11, mins=10, secs=59.

10. Истинно ли следующее утверждение: вы можете присвоить значение одной структурной переменной другой структурной переменной того же типа?

11. Напишите выражение, присваивающее переменной temp значение поля paw структурной переменной fido.

12. Перечисление объединяет:

    а) данные различных типов;

    б) логически связанные переменные;

    в) именованные целые числа;

    г) константные значения.

13. Напишите оператор, описывающий перечисление с именем players и набором значений B1, B2, SS, B3, RF, CF, LF, P и C.

14. Считая, что перечисление players задано так, как указано в вопросе 13, определите переменные joe и tom типа players и присвойте им значения LF и P соответственно.

15. Учитывая выражения, созданные при ответах на вопросы 13 и 14, укажите, какие из следующих операторов являются корректными:

    а) joe = QB;

    б) tom = SS;

    в) LF = tom;

    г) difference = joe – tom.

16. Первые три константы перечисляемого типа обычно представляются числами ___, ___ и ___.

17. Напишите оператор, в котором объявляется перечисляемый тип speeds с константами obsolete, single и album. Присвойте этим константам целые значения 78, 45 и 33.

18. Объясните, почему объявление

    ```
    Enum isWord{ NO, YES };
    ```

    более удачно, чем объявление

    ```
    Enum isWord{ YES, NO };
    ```

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Номер телефона, например (212) 767–8900, можно условно разделить на три части: код города (212), номер телефонной станции (767) и номер абонента (8900). Напишите программу с использованием структуры, позволяющую раздельно хранить эти три части телефонного номера. Назовите структуру phone. Создайте две структурные переменные типа phone. Инициализацию одной из них произведите сами, а значения для другой запросите с клавиатуры. Затем выведите содержимое обеих переменных на экран. Результат работы программы должен выглядеть приблизительно так:

    ```
    Введите код города, номер станции и номер абонента:
    415 555 1212
    Мой номер (212) 767-8900
    Ваш номер (415) 555-1212
    ```

*2. Расположение точки на плоскости можно задать с помощью двух координат: x и y. Например, если точка имеет координаты (4, 5), то это значит, что она отстоит на 4 единицы справа от вертикальной оси и на 5 единиц вверх

от горизонтальной оси. Сумма двух точек определяется как точка, имеющая координаты, равные сумме соответствующих координат слагаемых.

Напишите программу, использующую для интерпретации точки на плоскости структуру с названием point. Определите три переменные типа point, и две из них инициализируйте с помощью значений, вводимых с клавиатуры. Затем присвойте третьей переменной значение суммы первых двух переменных и выведите результат на экран. Результат работы программы может выглядеть следующим образом:

```
Введите координаты точки p1: 3 4
Введите координаты точки p2: 5 7
Координаты точки p1+p2 равны 8, 11
```

*3. Создайте структуру с именем Volume, содержащую три поля типа Distance из примера englstrc, для хранения трех измерений помещения. Определите переменную типа Volume, инициализируйте ее, вычислите объем, занимаемый помещением, и выведите результат на экран. Для подсчета объема переведите каждое из значений типа Distance в значение типа float, равное соответствующей длине в футах и хранимое в отдельной переменной. Затем для вычисления объема следует перемножить три полученные вещественные переменные.

4. Создайте структуру с именем employee, содержащую два поля: номер сотрудника типа int и величину его пособия в долларах типа float. Запросите с клавиатуры данные о трех сотрудниках, сохраните их в трех структурных переменных типа employee и выведите информацию о каждом из сотрудников на экран.

5. Создайте структуру типа date, содержащую три поля типа int: месяц, день и год. Попросите пользователя ввести день, месяц и год в формате 31/12/2002, сохраните введенное значение в структурной переменной, а затем извлеките данные из этой переменной и выведите их на экран в том же формате, в каком они вводились.

6. Как мы говорили, стандартные средства ввода/вывода C++ вместо значений перечисляемых типов данных выводят их внутреннее представление в виде целых чисел. Для того чтобы преодолеть это ограничение, вы можете использовать конструкцию switch, с помощью которой устанавливается соответствие между значением переменной перечисляемого типа и ее внутренним представлением. Пусть, например, в программе определен перечисляемый тип данных etype, отражающий должность сотрудника:

```
enum etype { laborer, secretary, manager, accountant, executive, researcher };
```

Напишите программу, которая сначала по первой букве должности, введенной пользователем, определяет соответствующее значение переменной, помещает это значение в переменную типа etype, а затем выводит полностью название должности, первую букву которой ввел пользователь. Взаимодействие программы с пользователем может выглядеть следующим образом:

```
Введите первую букву должности
(laborer, secretary, manager, accountant,
```

```
executive, researcher): a
Полное название должности: accountant
```

Возможно, вам понадобится два ветвления switch: одно — для ввода значения, другое — для вывода.

7. Добавьте поля типа enum etype (см. упражнение 6) и struct date (см. упражнение 5) в структуру employee из упражнения 4. Организуйте программу таким образом, чтобы пользователь вводил 4 пункта данных о каждом из трех сотрудников: его номер, величину зарплаты, его должность и дату принятия на работу. Программа должна хранить введенные значения в трех переменных типа employee и выводить их содержимое на экран.

8. Вернитесь к упражнению 9 главы 2 «Основы программирования на C++». В этом упражнении требуется написать программу, которая хранит значения двух дробей в виде числителя и знаменателя, а затем складывает эти дроби согласно арифметическому правилу. Измените эту программу так, чтобы значения дробей хранились в структуре fraction, состоящей из двух полей типа int, предназначенных для хранения числителя и знаменателя. Все значения дробей должны храниться в переменных типа fraction.

9. Создайте структуру с именем time. Три ее поля, имеющие тип int, будут называться hours, minutes и seconds. Напишите программу, которая просит пользователя ввести время в формате часы, минуты, секунды. Можно запрашивать на ввод как три значения сразу, так и выводить для каждой величины отдельное приглашение. Программа должна хранить время в структурной переменной типа time и выводить количество секунд в введенном времени, определяемое следующим образом:

```
long totalsecs = t1.hours*3600 + t1.minutes*60 + t1.seconds
```

10. Создайте структуру с именем sterling, хранящую денежные суммы в старой английской системе, описанной в упражнениях 8 и 11 главы 3 «Циклы и ветвления». Поля структуры могут быть названы pounds, shillings и pence и иметь тип int. Программа должна запрашивать у пользователя значение денежной суммы в новых десятичных фунтах (значение должно храниться в переменной типа double), затем переводить эту сумму в старую систему, сохранять переведенное значение в переменной типа sterling и выводить на экран полученную сумму в фунтах, шиллингах и пенсах.

11. Используя структуру time из упражнения 9, напишите программу, которая получает от пользователя два значения времени в формате 12:59:59, сохраняет их в переменных типа struct time, затем переводит оба значения в секунды, складывает их, переводит сумму в исходный формат, сохраняет его в переменной типа time и выводит полученный результат на экран в формате 12:59:59.

12. Переработайте программу-калькулятор для дробей, описанную в упражнении 12 главы 3 так, чтобы каждая из дробей хранилась как значение переменной типа struct fraction, по аналогии с упражнением 8 этой главы.

Глава 5
Функции

- ✦ Простые функции
- ✦ Передача аргументов в функцию
- ✦ Значение, возвращаемое функцией
- ✦ Ссылки на аргументы
- ✦ Перегруженные функции
- ✦ Рекурсия
- ✦ Встраиваемые функции
- ✦ Аргументы по умолчанию
- ✦ Область видимости и класс памяти
- ✦ Возвращение значения по ссылке
- ✦ Константные аргументы функции

Функция представляет собой именованное объединение группы операторов. Это объединение может быть вызвано из других частей программы.

Наиболее важной причиной использования функций служит необходимость концептуализировать структуру программы. Как мы уже упоминали в главе 1 «Общие сведения», деление программы на функции является базовым принципом структурного программирования (однако объектно-ориентированный подход является более мощным принципом организации программы).

Причиной, из-за которой в свое время была создана концепция функций, стало стремление сократить размер программного кода. Любая последовательность операторов, встречающаяся в программе более одного раза, будучи вынесенной в отдельную функцию, сокращает размер программы. Несмотря на то, что функция в процессе выполнения программы исполняется не один раз, ее код хранится только в одной области памяти. На рис. 5.1 показано, каким образом функция может вызываться из разных участков программы.

Функции в C++ схожи с подзадачами и процедурами других языков программирования.

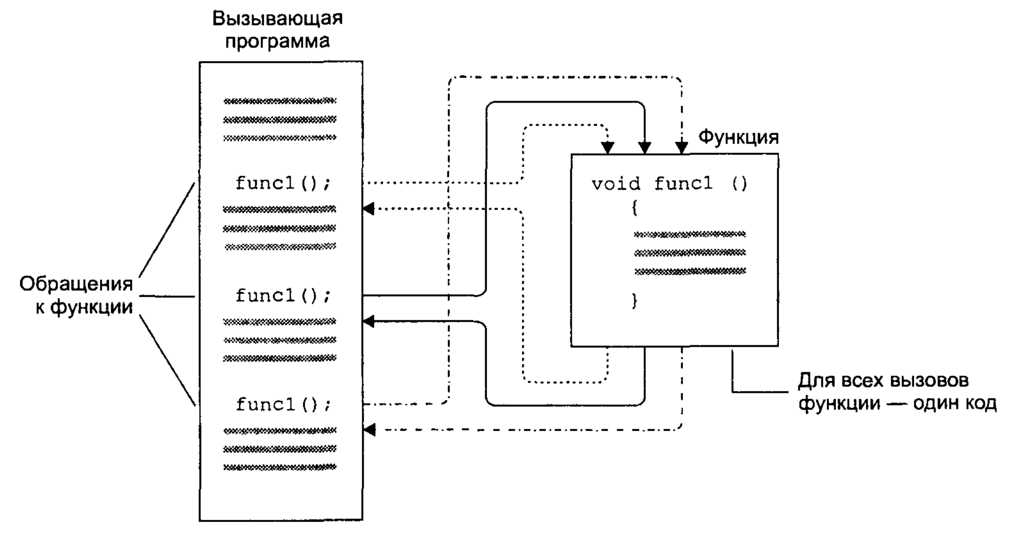


Рис. 5.1. Управление вызовами функции

Простые функции

Наш первый пример демонстрирует простую функцию, которая печатает строку из 45 символов *. В программе создается таблица, и для того, чтобы сделать эту таблицу удобочитаемой, используются разделяющие линии, состоящие из символов *.

Листинг программы TABLE выглядит следующим образом[1]:

```
// table.cpp
// demonstrates simple function
#include <iostream>
using namespace std;
void starline();              // объявление функции (прототип)
int main()
{
  starline();                 // вызов функции
  cout << "Тип данных   Диапазон" << endl;
  starline();                 // вызов функции
  cout << "char        -128...127" << endl
  << "short      -32.768...32.767" << endl
  << "int        Системно-зависимый" << endl
  << "long    -2.147.483.648...2.147.483.647" << endl;
  starline();                 // вызов функции
  return 0;
}
//--------------------------------------------------------
//определение функции starline()
void starline()               // заголовок функции
{
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
  for( int j=0; j<45; j++ ) // тело функции
  cout << '*';
  cout << endl;
}
```

Результат работы программы имеет следующий вид:

```
*********************************************
Тип данных Диапазон
*********************************************
char -128...127
short-32,768...32,767
intСистемно-зависимый
long-2,147,483,648...2,147,483,647
*********************************************
```

Программа включает в себя две функции: main() и starline(). Вы уже сталкивались с множеством примеров, в которых присутствовала всего одна функция — main(). Что необходимо добавить в программу для использования дополнительных функций? Обязательными являются три компонента: *объявление* функции, ее *определение* и *вызовы*.

Объявление функции

Подобно тому как вы не можете использовать переменную, не сообщив компилятору информацию о ней, вы не можете обратиться к функции, не указав в программе ее необходимые атрибуты. Есть два способа описать функцию: *объявить* или *определить* функцию до первого ее вызова. Здесь мы рассмотрим только объявление функции, а определение функции будет рассмотрено позднее. В программе TABLE объявление функции starline() выглядит следующим образом:

```
void starline();
```

Объявление функции означает, что где-то ниже в листинге программы будет содержаться код этой функции. Ключевое слово void указывает на то, что функция не возвращает значения, а пустые скобки говорят об отсутствии у функции передаваемых аргументов (для того чтобы явно показать отсутствие аргументов у функции, вы можете поместить внутрь скобок слово void; такая практика часто применяется в языке С, однако в С++ чаще оставляют скобки пустыми). Мы более подробно рассмотрим аргументы функции и ее возвращаемое значение чуть позже.

Обратите внимание на то, что объявление функции заканчивается точкой с запятой (;) и на самом деле является обычным оператором.

Объявления функций также называют *прототипами* функций, поскольку они являются неким общим представлением или описанием функций. Прототип говорит компилятору о том, что «функция, имеющая данные атрибуты, будет написана позже, и можно вызывать эту функцию до того, как будет обнаружен ее код». Информацию о функции, содержащуюся в ее объявлении (тип возвращаемого значения, а также число и типы аргументов функции) также иногда называют *сигнатурой* функции.

Вызов функции

Функция starline() трижды вызывается из функции main(). Все три вызова выглядят одинаково:

```
starline();
```

Для того чтобы вызвать функцию, нам понадобились только имя функции и круглые скобки. Вызов функции внешне очень похож на прототип; разница заключается лишь в том, что при вызове не указывается тип возвращаемого значения. Вызов функции завершается точкой с запятой (;). Выполнение оператора вызова функции инициирует выполнение самой функции. Это означает, что управление передается операторам функции, которые после своего выполнения, в свою очередь, передают управление оператору, следующему за вызовом функции.

Определение функции

Теперь мы подошли к рассмотрению самой функции, или ее определения. Определение содержит код функции. Для функции starline() определение выглядит следующим образом:

```
void starline()              // заголовок функции
{
  for( int j=0; j<45; j++ )   // тело функции
    cout << '*';
  cout << endl;
}
```

Определение функции состоит из *заголовка* и *тела* функции. Тело функции состоит из последовательности операторов, заключенной в фигурные скобки. Заголовок функции должен соответствовать ее прототипу: имя функции и тип возвращаемого ей значения должны совпадать с указанными в прототипе; кроме того, аргументы функции, если они есть, должны иметь те же типы и следовать в том же порядке, в каком они указывались в прототипе.

Обратите внимание на то, что заголовок функции не ограничивается точкой с запятой (;). На рисунке 5.2 продемонстрирован синтаксис объявления, вызова и определения функции.

Когда происходит вызов функции, программа передает управление первому оператору тела функции. Затем исполняются операторы, находящиеся в теле функции, и когда достигается закрывающая фигурная скобка, управление передается обратно вызывающей программе.

В табл. 5.1 сведена информация о трех компонентах функции.

Таблица 5.1. Компоненты функции

Название	Назначение	Пример
Объявление (прототип)	Содержит имя функции, типы ее аргументов и возвращаемого значения. Указывает компилятору на то, что определение функции будет сделано позднее	void func();

продолжение ⇨

Таблица 5.1 (*продолжение*)

Название	Назначение	Пример
Вызов	Указывает на то, что необходимо выполнить функцию	func();
Определение	Является собственно функцией. Содержит код, предназначенный для исполнения	void func() { // операторы }
Заголовок	Первая строка определения	void func()

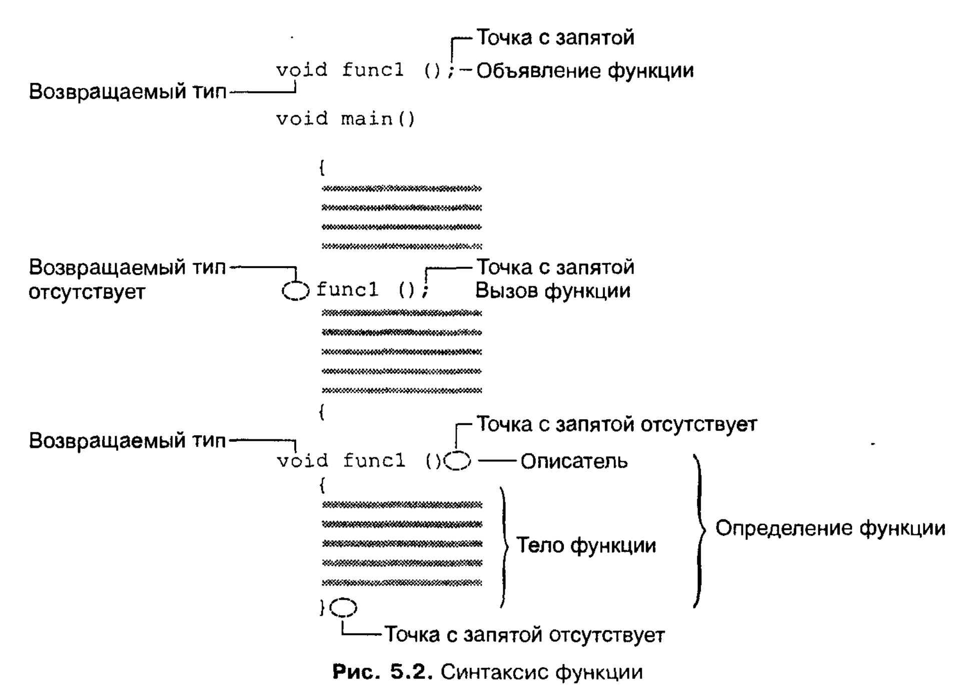


Рис. 5.2. Синтаксис функции

Обычные и библиотечные функции

Мы уже использовали в наших примерах библиотечные функции. В кодах наших программ встречались вызовы библиотечных функций, например

```
ch = getche();
```

Объявления библиотечных функций содержатся в заголовочных файлах, подключаемых к программе (в случае функции getche() таким файлом является CONIO.H). Определение, уже скомпилированное в исполняемый код, находится в библиотечном файле, содержимое которого автоматически прикомпоновывается к исполняемому коду программы.

Если мы используем библиотечную функцию, нам не нужно самим создавать ее объявление и определение, но в том случае, когда мы разрабатываем свою собственную функцию, и объявление, и определение этой функции должны при-

сутствовать в исходном тексте программы. В программе TABLE мы можем наблюдать это на примере функции starline() (процесс создания функций усложняется в программах, состоящих из нескольких файлов, о чем пойдет речь в главе 13 «Многофайловые программы»).

Отсутствие объявления

Вторым способом вставить свою функцию в программу является ее определение, помещенное ранее первого ее вызова. В этом случае прототип функции не используется. Мы можем переработать программу TABLE таким образом, что определение функции starline() будет располагаться до первого ее вызова. Полученную программу назовем TABLE2.

```
// table2.cpp
// определение функции без предварительного использования
// прототипа
#include <iostream>
using namespace std.
//--------------------------------------------------------
// определение функции starline()
void starline()
{
  for( int j=0: j<45 :j++ )
  cout << '*':
  cout << endl;
}
//--------------------------------------------------------
int main()                    // функция main()
{
  starline():                 // вызов функции
  cout << "Тип данных  Диапазон" << endl;
  starline();                 // вызов функции
  cout << "char        -128...127" << endl
  << "short      -32.768...32.767" << endl
  << "int        Системно-зависимый" << endl
  << "long    -2.147.483.648.  2.147.483.647" << endl;
  starline():                 // вызов функции
  return 0:
}
```

Такой подход более удобен для применения в коротких программах, поскольку не требует указания прототипа функции. Однако вместе с этим теряется и гибкость работы с функциями. Если число используемых функций велико, то программист будет вынужден следить за тем, чтобы тело каждой из функций располагалось раньше, чем любой ее вызов из других функций. Иногда эта задача бывает практически невыполнимой. Кроме того, некоторые программисты предпочитают располагать функцию main() первой в листинге, поскольку она всегда выполняется первой. Обычно мы будем оформлять программу именно таким образом: использовать прототипы функций и начинать листинг с функции main().

Передача аргументов в функцию

Аргументом называют единицу данных (например, переменную типа int), передаваемую программой в функцию. Аргументы позволяют функции оперировать различными значениями или выполнять различные действия в зависимости от переданных ей значений.

Передача констант в функцию

В качестве примера рассмотрим следующую ситуацию. Пусть нам необходимо сделать функцию starline() более гибкой: мы хотим изменить ее таким образом, чтобы она выводила не фиксированное (равное 45 в предыдущих примерах), а любое заданное количество символов. В следующем примере, TABLEARG, мы создаем модификацию функции starline() под названием repchar(). В качестве аргументов для функции используется символ, предназначенный для печати, а также число раз, которое данный символ будет выведен на экран.

```
// tablearg.cpp
// демонстрирует передачу аргументов в функцию
#include <iostream>
using namespace std;
void repchar(char, int);        // прототип функции
int main()
{
  repchar('-', 43);             // вызов функции
  cout << "Тип данных  Диапазон" << endl;
  repchar('=', 23);             // вызов функции
  cout << "char        -128...127" << endl
  << "short      -32.768...32.767" << endl
  << "int        Системно-зависимый " << endl
  << "double     -2.147.483.648...2.147.483.647" << endl;
  repchar('-', 43);             // вызов функции
  return 0;
}
//--------------------------------------------------------------
// определение функции repchar()
void repchar(char ch, int n)    // заголовок функции
{
  for( int j=0; j<n; j++ )      // тело функции
  cout << ch;
  cout << endl;
}
```

Итак, новая функция называется repchar(). Ее прототип выглядит следующим образом:

```
void repchar(char, int); //объявление с указанием типов аргументов
```

В скобках указаны типы данных, которые будут иметь передаваемые в функцию аргументы: char и int.

При вызове функции вместо аргументов в скобках указываются их конкретные значения (в данном случае константы):

```
repchar('-', 43);          // в вызове функции используются значения аргументов
```

Результатом данного вызова будет печать 43 символов '-'. При указании значений аргументов важно соблюдать порядок их следования: сначала должно быть указано значение типа char, соответствующее символу, выводимому на экран, а затем значение типа int, которое определяет число раз, которое указанный символ будет напечатан. Кроме того, типы аргументов в объявлении и определении функции также должны быть согласованы.

Вызов функции repchar()

```
repchar('='. 23);
```

приведет к появлению на экране 23 знаков равенства. Третий вызов снова выведет на печать 43 знака '-'. Результат работы программы TABLEARG будет выглядеть следующим образом:

```
-------------------------------------------
Тип данных Диапазон
====================
char-128 to 127
short-32.768 to 32.767
intСистемно-зависимый
long-2.147.483.648 to 2.147.483.647
-------------------------------------------
```

Программа передает в функцию аргументы, например '-' и 43. Переменные, используемые внутри функции для хранения значений аргументов, называются *параметрами*. В функции repchar() в качестве параметров выступают переменные ch и n (необходимо отметить, что многие программисты используют термины *аргумент* и *параметр* как эквивалентные). При определении функции в ее заголовке мы указываем типы и имена параметров:

```
void repchar(char ch. int n)    // в прототипе указаны типы и имена параметров
```

Параметры используются внутри функции так же, как обычные переменные. Присутствие имен n и ch в прототипе функции repchar() эквивалентно паре операторов

```
char ch:
char n;
```

Когда вызов функции произведен, параметры автоматически инициализируются значениями, переданными программой.

Передача значений переменных в функцию

В примере TABLEARG роль аргументов выполняли константы: '-', 43 и т. д. Теперь давайте рассмотрим пример, в котором вместо констант в функцию будут передаваться значения переменных. Следующая программа, VARARG, включает в себя ту же функцию repchar(), что и программа TABLEARG, но теперь символ и число его повторений будут задаваться пользователем:

```
// vararg.cpp
// передача переменных в функцию в качестве аргументов
```

```
#include <iostream>
using namespace std;
void repchar(char, int);        // объявление функции
int main()
{
  char chin;
  int nin;
  cout << "Введите символ: ";
  cin >> chin;
  cout << "Введите число повторений символа: ";
  cin >> nin;
  repchar(chin, nin);
  return 0;
}
//--------------------------------------------------------
// определение функции repchar()
void repchar(char ch, int n)    // заголовок функции
{
  for(int j=0; j<n; j++)        // тело функции
  cout << ch;
  cout << endl;
}
```

Пример взаимодействия программы VARARG с пользователем может выглядеть так:

```
Введите символ: +
Введите число повторений символа: 20
++++++++++++++++++++
```

В этой программе переменные chin и nin функции main() передаются в качестве аргументов в функцию repchar():

```
repchar(chin, nin);             // вызов функции
```

Типы переменных, используемых в качестве аргументов функции, должны, как и в случае констант, совпадать с типами, указанными в объявлении и определении функции. В данном случае это требование сводится к тому, чтобы переменная chin имела тип char, а переменная nin имела тип int.

Передача аргументов по значению

В программе VARARG в момент выполнения вызова функции значения переменных chin и nin будут переданы функции. Как и в случае с константами, функция имеет две переменные для хранения переданных значений. Типы и имена этих переменных указаны в прототипе при определении функции: char ch и int n. Переменные ch и n инициализируются переданными в функцию значениями, после чего доступ к ним осуществляется так же, как и к другим переменным функции.

Способ передачи аргументов, при котором функция создает копии передаваемых значений, называется *передачей аргументов по значению*. В этой главе мы также рассмотрим другой способ передачи значений в функцию — по *ссылке*. На рис. 5.3 показано, каким образом происходит создание новых переменных в случае передачи аргументов по значению.

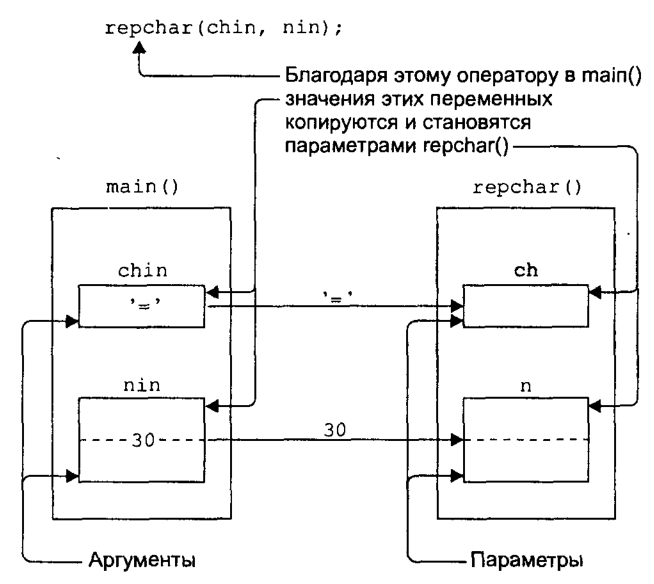


Рис. 5.3. Передача по значению

Структурные переменные в качестве аргументов

Структурные переменные могут использоваться в качестве аргументов функций. Мы продемонстрируем два примера, в одном из которых участвует уже знакомая нам структура Distance, а другой пример будет работать с графическим объектом.

Структура Distance

Следующий пример, ENGLDISP, содержит функцию, которая использует аргумент типа Distance — структуры, фигурировавшей в нескольких примерах из главы 4 «Структуры».

```
// engldisp.cpp
// передача структурных переменных в функцию
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
struct Distance              // длина в английской системе
{
  int feet;
  float inches;
};
//////////////////////////////////////////////////////////
void engldisp( Distance );   // объявление
int main()
{
  Distance d1,d2;            // определение двух длин
                             // ввод значений полей d1
  cout << "Введите число футов: "; cin >> d1.feet;
  cout << "Введите число дюймов: "; cin >> d1.inches;
                             // ввод значений полей d2
```

```
  cout << "\nВведите число футов: "; cin >> d2.feet;
  cout << "Введите число дюймов: "; cin >> d2.inches;
  cout << "\nd1 = ";
  engldisp(d1);              // вывод значения d1
  cout << "\nd2 = ";
  engldisp(d2);              // вывод значения d2
  cout << endl;
  return 0;
}
//--------------------------------------------------------
// функция engldisp(), отображающая значения
// полей структурной переменной типа Distance
void engldisp(Distance dd )  // параметр dd типа Distance
{
  cout << dd.feet << "\'-" << dd inches << "\"";
}
```

В функции main() производится запрос на ввод пользователем двух расстояний, каждое из которых задается числом футов и числом дюймов. Полученные значения сохраняются в двух структурных переменных d1 и d2. Затем производится вызов функции engldisp(), принимающей в качестве аргумента значение типа Distance. Задачей функции является вывод расстояния в стандартной форме записи, такой, как 10'-2.25". Результат взаимодействия программы с пользователем выглядит следующим образом:

```
Введите количество футов. 9
Введите количество дюймов: 4
Введите количество футов: 5
Введите количество дюймов: 4.25
d1 = 6'-4"
d2 = 5'-4.25"
```

При объявлении функции, ее определении и вызовах, структурная переменная использовалась в качестве аргумента так же, как и любая другая переменная стандартного типа char или int.

В функции main() обращение к функции engldisp() происходит дважды. В первом случае в качестве аргумента передается переменная d1, во втором случае — переменная d2. Параметром функции engldisp() служит переменная dd типа Distance. Как и в случаях с переменными стандартных типов, параметру dd присваивается значение передаваемой функцией main() величины типа Distance. Операторы функции engldisp() могут получать доступ к полям переменной dd с помощью выражений dd.feet и dd.inches. На рис. 5.4 показано, каким образом структурная переменная передается в качестве аргумента функции.

Как в случаях с простыми переменными, структурный параметр dd является лишь копией аргументов d1 и d2, передаваемых в функцию. Поэтому если бы функция engldisp() изменяла значение переменной dd (хотя в данном примере она не делает этого) с помощью операторов

```
dd.feet = 2;
dd.inches = 3.25;
```

то сделанные изменения никак не отразились бы на значениях переменных d1 и d2.

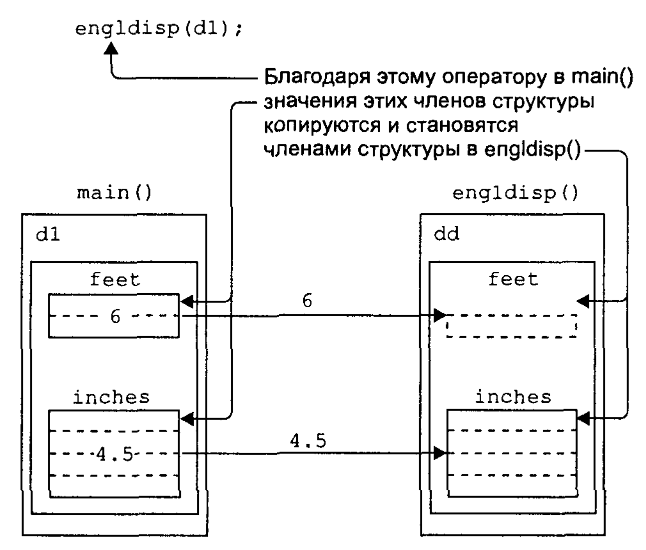


Рис. 5.4. Передача структуры в качестве аргумента

Структура circle

Следующий пример передачи структурной переменной в функцию будет реализован с использованием *консольной графики*. Исходные и заголовочные файлы для этих функций перечислены в приложении Д «Упрощенный вариант консольной графики». При необходимости их можно найти на сайте издательства, адрес которого указан во введении к этой книге. Вам нужно будет включить соответствующие заголовочные файлы MSOFTCON.H или BORLACON.H и исходные файлы MSOFTCON.CPP или BORLACON.CPP в проект. Функции консольной графики описаны в приложении Д, а процедура добавления указанных файлов к проекту — в приложениях В «Microsoft Visual C++» и Г «Borland C++ Builder».

В этом примере структура с именем circle представляет графический объект типа «круг». Круги имеют определенные координаты своего центра на экране консоли и радиус. Кроме того, они обладают определенным цветом и стилем заполнения, информацию о которых можно найти в приложении Д. Приведем листинг программы CIRCSTRC:

```
// circstrc.cpp
// графические объекты типа круг
#include "msoftcon.h"           // для графических функций
////////////////////////////////////////////////////////////
struct circle                   // графический объект "круг"
{
  int xCo, yCo;                 // координаты центра
  int radius;
  color fillcolor;              // цвет
  fstyle fillstyle;             // стиль заполнения
};
////////////////////////////////////////////////////////////
void circ_draw(circle c)
```

```
{
  set_color(c.fillcolor);       // установка цвета
  set_fill_style(c.fillstyle);  // установка стиля заполнения
  draw_circle(c.xCo,c.yCo,c.radius); // рисование круга
}
//--------------------------------------------------------
int main()
{
  init_graphics();              // инициализация графики
  // создание кругов
  circle c1 = {15, 7, 5, cBLUE, X_FILL };
  circle c2 = {41, 12, 7, cRED, O_FILL };
  circle c3 = {65, 18, 4, cGREEN, MEDIUM_FILL };
  circ_draw(c1);                // рисование кругов
  circ_draw(c2);
  circ_draw(c3);
  set_cursor_pos(1, 25);        //cursor to lower left corner
  return 0;
}
```

Каждая из переменных c1, c2 и c3 типа circle инициализируется своим набором значений полей. Для c1 это выглядит следующим образом:

```
circle c1 = { 15, 7, 5, cBlue, X_FILL };
```

Предполагается, что ваша консоль имеет размер 80 колонок на 25 строк. Число 15 означает номер колонки (координату x), а число 7 — номер строки (координата y, отсчитываемая от верхней границы экрана), определяющие местоположение центра круга. Далее, число 5 задает радиус окружности, cBLUE — цвет круга, а X_FILL означает, что круг будет заполнен символами 'X'. Аналогичным образом задаются параметры остальных кругов.

Когда параметры для всех кругов заданы, мы рисуем их на экране путем троекратного вызова функции circ_draw() — по одному вызову для каждого из кругов. На рис. 5.5 приведен результат работы программы CIRCSTRC. Можно заметить, что круги выглядят несколько неровно — это является следствием небольшого числа единиц отображения на экране в консольном режиме.

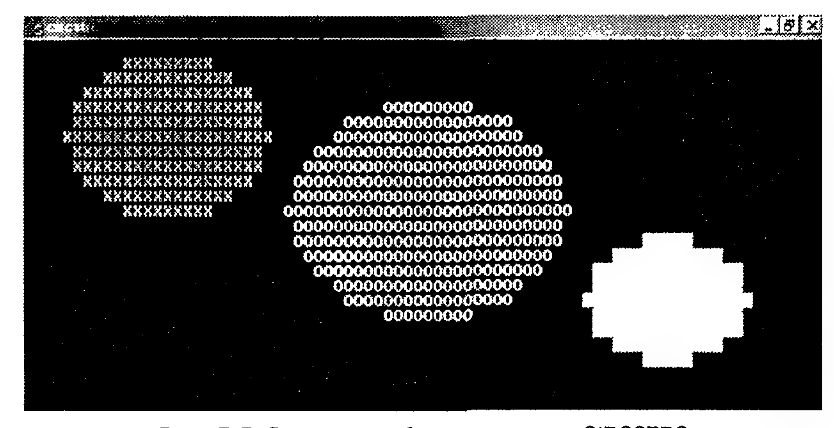


Рис. 5.5. Результат работы программы CIRCSTRC

Обратите внимание на то, что структуры хранят параметры кругов, в то время как функция circ_draw() заставляет эти круги рисовать себя на экране. Как мы выясним в главе 6 «Объекты и классы», объекты образуются путем объединения структур и функций в единые элементы, способные как хранить данные, так и выполнять действия над ними.

Имена переменных внутри прототипа функции

Существует способ сделать прототипы ваших функций проще для восприятия. Суть способа заключается в размещении имен параметров в прототипе вместе со своими типами. Рассмотрим такой пример. Пусть ваша функция изображает точку на экране. Вы могли бы записать ее прототип обычным способом, как мы делали раньше:

```
void display_point(int. int);             // объявление функции
```

однако более наглядным было бы следующее представление:

```
void display_point(int horiz. int vert); //объявление функции
```

Эти два объявления функции абсолютно одинаково воспринимаются компилятором, но в первом случае прототип не несет информации о том, какой из аргументов будет играть роль абсциссы, а какой — ординаты. Преимущество второго из прототипов заключается в том, что, глядя на него, можно легко понять, где находится абсцисса, а где — ордината, а это снижает вероятность ошибки при вызове функции.

Обратите внимание на то, что имена, указываемые в прототипе функции, никак не связаны с именами переменных, передаваемых в качестве аргументов при вызове функции:

```
display_point(x. y);            // вызов функции
```

В наших программах мы часто будем использовать имена параметров в прототипах функций.

Значение, возвращаемое функцией

Когда выполнение функции завершается, она может возвратить значение программе, которая ее вызвала. Как правило, возвращаемое функцией значение имеет отношение к решению задачи, возложенной на эту функцию. Следующий пример демонстрирует применение функции, получающей значение веса в фунтах и возвращающей эквивалентное значение этого веса в килограммах:

```
// convert.cpp
// демонстрирует механизм возврата значения функцией
#include <iostream>
using namespace std:
float lbstokg(float);            // прототип функции
int main()
```

```
{
  float lbs, kgs;
  cout << "\nВведите вес в фунтах: ";
  cin >> lbs;
  kgs = lbstokg(lbs);
  cout << "Вес в килограммах равен " << kgs << endl;
  return 0;
}
//--------------------------------------------------------
// функция lbstokg()
// переводит фунты в килограммы
float lbstokg( float pounds )
{
  float kilograms = 0.453592 * pounds;
  return kilograms;
}
```

Результат работы программы может выглядеть следующим образом:

```
Введите вес в фунтах: 182
Вес в килограммах равен 82.553741
```

В случае, если функция возвращает значение, тип этого значения должен быть определен. Тип возвращаемого значения указывается перед именем функции при объявлении и определении функции. В нашем последнем примере функция lbstokg() возвращает значение типа float, что отражено в ее объявлении:

```
float lbstokg(float);
```

Первое слово float означает, что функция lbstokg() возвращает значение типа float, а слово float, заключенное в скобках, указывает на наличие у функции lbstokg() одного аргумента типа float.

В более ранних наших примерах функции не возвращали значения, поэтому перед именами таких функций вместо названия типа находилось ключевое слово void.

Если функция возвращает значение, то вызов функции рассматривается как выражение, значение которого равно величине, возвращаемой функцией. Мы можем использовать это выражение аналогично любым другим выражениям; например, мы можем присвоить его значение переменной:

```
kgs = lbstokg(lbs);
```

Здесь переменной kgs присваивается значение, возвращенное функцией lbstokg().

Оператор return

Функция lbstokg() получает в качестве аргумента значение веса, выраженное в фунтах, которое хранится в параметре pounds. Эквивалентный вес в килограммах вычисляется путем умножения переменной pounds на константу и записывается в переменную kilograms. Значение переменной kilograms затем возвращается программе с помощью оператора

```
return kilograms;
```

Обратите внимание на то, что значение веса в килограммах хранится как в функции lbstokg(), так и в функции main(), соответственно в переменных kilograms и kgs. В момент возвращения функцией значения происходит копирование значения переменной kilograms в переменную kgs. Программа может получить значение переменной kilograms только через механизм возврата значения; доступ к самой переменной kilograms из программы невозможен. Все вышесказанное проиллюстрировано на рис. 5.6.

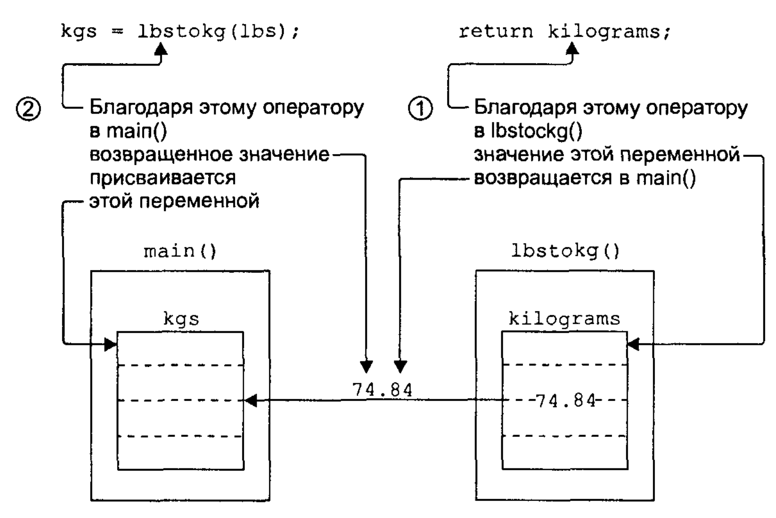


Рис. 5.6. Возврат значения

Количество аргументов у функции может быть сколь угодно большим, но возвращаемое значение всегда только одно. Эта особенность функций является препятствием для нас в тех случаях, когда нам необходимо вернуть программе несколько значений, однако есть способы, позволяющие возвращать и несколько значений при помощи функций. Одним из таких способов, который мы рассмотрим в этой главе, является *передача аргументов по ссылке*. Другой способ — вернуть структурную переменную, в полях которой будут располагаться нужные значения.

Всегда следует указывать тип значения, возвращаемого функцией. Если ваша функция не возвращает значения, то вместо типа возвращаемого значения должно присутствовать ключевое слово void. Если же вы не укажете возвращаемый тип данных при объявлении функции, то по умолчанию возвращаемым типом будет int. Например, прототип

```
somefunc();          // подразумевается возвращаемый тип int
```

указывает на то, что функция somefunc() возвращает значение типа int.

Это обусловлено причинами, имеющими корни в ранних версиях языка С. Однако на практике не следует использовать тип, возвращаемый по умолчанию. Лучше явно указывать возвращаемый тип даже в том случае, если этим типом является int. Это сделает ваш листинг более понятным и легко читаемым.

Исключение ненужных переменных

Программа CONVERT содержит несколько переменных, употребление которых обусловлено исключительно соображениями наглядности и простоты кода. Следующая программа, CONVERT2, являющаяся модификацией программы CONVERT, демонстрирует, каким образом можно использовать в выражениях функции вместо переменных:

```
// convert2.cpp
// более компактная версия программы convert
#include <iostream>
using namespace std;
float lbstokg(float);          // прототип функции
int main()
{
  float lbs;
  cout << "\nВведите вес в фунтах: ";
  cin >> lbs;
  cout << "Вес в килограммах равен " << lbstokg(lbs)
  << endl;
  return 0;
}
//--------------------------------------------------------
// функция lbstokg()
// переводит фунты в килограммы
float lbstokg(float pounds)
{
  return 0.453592 * pounds;
}
```

В теле функции main() теперь отсутствует переменная kgs. Вместо этого вызов функции lbstokg(lbs) помещен внутрь оператора cout:

```
cout << "Вес в килограммах равен " << lbstokg(lbs) <<
endl;
```

В теле функции lbstokg() больше не используется переменная kilograms, а выражение 0.453592 * pounds помещено непосредственно в оператор return:

```
return 0.453592 * pounds;
```

Как только вычисление произведено, его результат возвращается программе так же, как это было сделано с использованием временной переменной.

Из соображений удобочитаемости программисты часто ставят скобки, окружающие входящее в состав оператора return выражение:

```
return (0.453592 * pounds);
```

Даже в том случае, когда компилятор не требует наличия скобок в выражении, они не влияют на процесс вычисления, но в то же время облегчают чтение листинга другими программистами.

Опытные программисты предпочтут более длинный код программы CONVERT «экономичному» коду программы CONVERT2, поскольку последний менее понятен, особенно для людей, неискушенных в программировании. Решение вопроса выбора между краткостью и простотой программного кода лежит на вас,

поскольку только вы можете предполагать, для какой аудитории будет предназначен тот или иной листинг.

Структурная переменная в качестве возвращаемого значения

Мы видели, что структурные переменные можно использовать в качестве аргументов функций. Оказывается, что такие переменные могут выступать и в качестве значения, возвращаемого функцией. Следующий пример, RETSTRC, включает в себя функцию, складывающую два значения типа Distance и возвращающую результат этого же типа:

```
// retstrc.cpp
// демонстрирует возвращение функцией структурной переменной
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
struct Distance                      // длина в английской системе
{
  int feet;
  float inches;
};
///////////////////////////////////////////////////////////
Distance addengl( Distance, Distance ); // прототипы
void engldisp(Distance);
int main()
{
  Distance d1, d2, d3;               // три структурные переменные
  // ввод значения d1
  cout << "\nВведите число футов: "; cin >> d1.feet;
  cout << "Введите число дюймов: "; cin >> d1.inches;
  // ввод значения d2
  cout << "\nВведите число футов: "; cin >> d2.feet;
  cout << "Введите число дюймов: "; cin >> d2.inches;
  d3 = addengl(d1, d2);              // d3 равно сумме d1 и d2
  cout << endl;
  engldisp(d1); cout << " + ";       // вывод всех длин
  engldisp(d2); cout << " = ";
  engldisp(d3); cout << endl;
  return 0;
}
//--------------------------------------------------------
// функция addengl()
// складывает два значения типа Distance и возвращает сумму
Distance addengl(Distance dd1,Distance dd2 )
{
  Distance dd3;                    // переменная для хранения будущей суммы
  dd3.inches = dd1.inches + dd2.inches; // сложение дюймов
  dd3.feet = 0;                    // с заемом;
  if(dd3.inches >= 12.0)           // если число дюймов больше 12.0,
  {                                // то уменьшаем число дюймов
    dd3.inches -= 12.0;            // на 12.0 и увеличиваем
    dd3.feet++;                    // число футов на 1
  }
```

```
  dd3.feet += dd1.feet + dd2.feet; // сложение футов
  return dd3;                      // возврат значения
}
//--------------------------------------------------------
// функция engldisp()
// отображает поля структурной переменной Distance
void engldisp(Distance dd)
{
  cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

Программа просит пользователя ввести две длины в виде числа футов и дюймов, затем складывает введенные значения путем вызова функции addengl() и выводит результат на экран с помощью функции engldisp(), взятой из программы ENGLDISP. Результат работы программы RETSTRC выглядит следующим образом:

```
Введите число футов: 4
Введите число дюймов: 5.5

Введите число футов: 5
Введите число дюймов: 6.5

4'-5.5" + 5'-6.5" = 10'-0"
```

Функция main() складывает две длины, хранящиеся в переменных типа Distance, с помощью вызова функции addengl():

```
d3 = addengl(d1, d2);
```

Функция addengl() возвращает сумму аргументов d1 и d2 в виде значения типа Distance, которое в функции main() присваивается структурной переменной d3.

Кроме структурного значения, возвращаемого функцией, в данной программе используются две функции, а не одна (если не считать функцию main()), как это было в предыдущих примерах. Функции программы могут располагаться в любом порядке. Единственным условием является появление прототипа каждой из функций до того, как производится первое обращение к данной функции.

Ссылки на аргументы

Ссылка является *псевдонимом*, или *альтернативным именем* переменной. Одним из наиболее важных применений ссылок является передача аргументов в функции.

Мы уже видели несколько примеров передачи аргументов в функции по значению. Когда осуществляется передача по значению, вызываемая функция создает новые переменные, имеющие те же типы, что и передаваемые аргументы, и копирует значения аргументов в эти переменные. Как мы видели, функция не имеет доступа к переменным-аргументам, а работает со сделанными ей копиями значений. Разумеется, такой механизм полезен в тех случаях, когда у функции нет необходимости изменять значения аргументов, и мы защищаем аргументы от несанкционированного доступа.

Передача аргументов по ссылке происходит по другому механизму. Вместо того чтобы передавать функции значение переменной, ей передается ссылка на эту переменную (фактически в функцию передается адрес переменной-аргумента в памяти, но пока эта деталь не столь важна для нас).

Важной особенностью передачи аргументов по ссылке является то, что функция имеет прямой доступ к значениям аргументов. К достоинствам ссылочного механизма также относится возможность возвращения функцией программе не одного, а множества значений.

Передача по ссылке аргументов стандартных типов

Следующий пример, REF, демонстрирует передачу по ссылке обычной переменной.

```
// ref.cpp
// применение ссылочного механизма передачи аргументов
#include <iostream>
using namespace std;
int main()
{
  void intfrac(float, float&, float&);      // прототип
  float number, intpart, fracpart;
  do {
    cout << "\nВведите вещественное число:";
    cin >> number;                          // ввод числа пользователем
    intfrac(number, intpart, fracpart);    // нахождение целой и дробной части
    cout << "Целая часть равна " << intpart // вывод результатов
    << ", дробная часть равна " << fracpart << endl;
  } while(number !=0.0 );                   //выход из цикла, если введен ноль
  return 0;
}
//--------------------------------------------------------
// функция intfrac()
// вычисляет целую и дробную часть вещественного числа
void intfrac(float n, float& intp, float& fracp)
{
  long temp = static_cast<long>(n);         // преобразование к типу long,
  intp = static_cast<float>(temp);          // и обратно во float
  fracp = n - intp;                         // вычитаем целую часть
}
```

Функция main() запрашивает у пользователя значение типа float. Затем программа выделяет целую и дробную часть введенного числа. Например, если пользователь введет число 12.456, то программой будет выдано сообщение о том, что целая часть введенного значения равна 12, а дробная часть равна 0.456. Для выделения целой части числа используется функция intfrac(). Вот пример работы программы:

```
Введите вещественное число: 99.44
Целая часть равна 99, дробная часть равна 0.44
```

Некоторые компиляторы могут сгенерировать ложные знаки после десятичной точки у дробной части числа, например 0.440002. Это является ошибкой работы компилятора, а не программы, поэтому можете смело проигнорировать их.

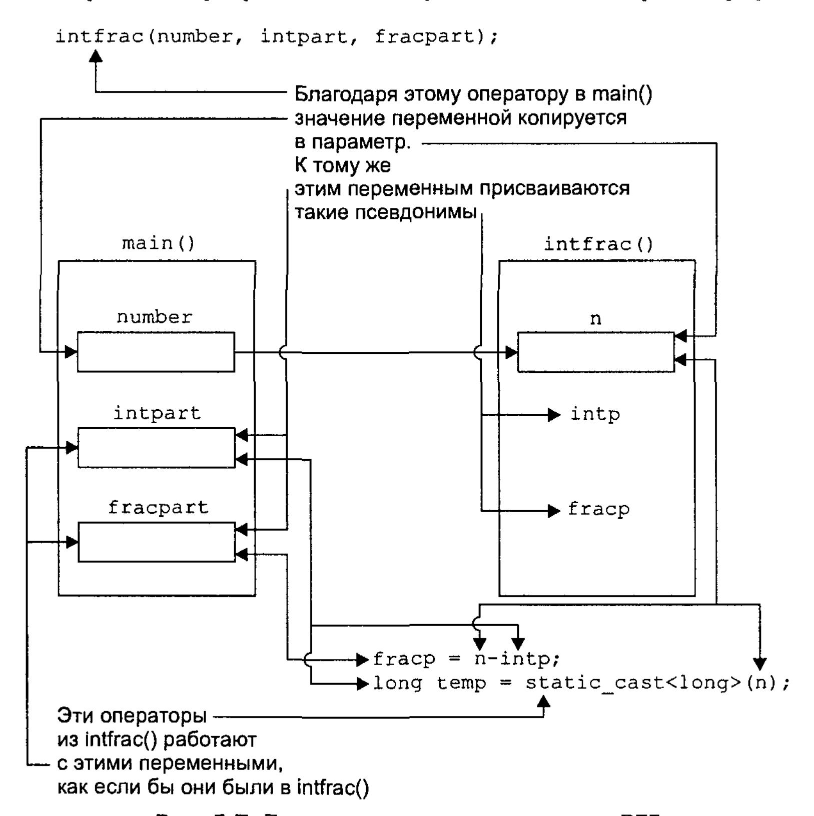


Рис. 5.7. Передача по ссылке в программе REF

Обратимся к рис. 5.7. Функция intfrac() выделяет из числа целую часть путем преобразования значения, переданного параметру n, в переменную типа long с использованием явного приведения типов:

```
long temp = static_cast<long>(n);
```

В результате выполнения такого оператора в переменной temp окажется целая часть введенного числа, поскольку при преобразовании вещественного значения к целому дробная часть будет отсечена. Затем целая часть снова преобразовывается в тип float:

```
intp = static_cast<float>(temp);
```

Дробная часть числа представляет собой разность между исходным числом и его целой частью (отметим, что существует библиотечная функция fmod(), выполняющая сходные действия над переменной типа double).

Функция intfrac() нашла целую и дробную часть числа, но каким образом она теперь сможет возвратить найденные значения в функцию main()? Ведь с помо-

щью оператора return можно возвратить только одно значение! Проблема решается путем использования ссылочного механизма. Прототип функции intfrac() выглядит следующим образом:

```
void intfrac(float n, float& intp, float& fracp)
```

Аргументы, передаваемые по ссылке, обозначаются знаком &, который следует за типом аргумента:

```
float& intp
```

Знак & означает, что intp является псевдонимом для той переменной, которая будет передана в функцию в качестве аргумента. Другими словами, используя имя intp в функции intfrac(), фактически мы оперируем значением переменной intpart функции main(). Знак & (амперсанд) символизирует ссылку, поэтому запись

```
float& intp
```

означает, что intp является ссылкой на переменную типа float. Точно так же fracp является ссылкой на переменную типа float, в нашем случае — на переменную fracpart.

Если какие-либо аргументы передаются в функцию по ссылке, то в прототипе функции необходимо с помощью знака & указывать соответствующий параметр-ссылку:

```
void intfrac(float, float&, float&);   // амперсанды
```

Аналогичным образом знаком & отмечаются ссылочные параметры в определении функции. Однако обратите внимание на то, что при вызовах функции амперсанды отсутствуют:

```
intfrac(number, intpart, fracpart);    //амперсанды отсутствуют
```

При вызове функции нет смысла сообщать, будет ли передан аргумент по значению или по ссылке.

В то время как переменные intpart и fracpart передаются в функцию по ссылке, переменная number передается по значению. Имена intp и intpart связаны с одной и той же областью памяти, так же как и пара имен fracp и fracpart. Параметр n не связан с переменной, передаваемой ему в качестве аргумента, поэтому ее значение копируется в переменную n. Поскольку не требуется изменять значение, хранящееся в переменной number, передача ее в функцию осуществляется по значению.

Программистов на C, вероятно, удивляет, что знак амперсанда используется одновременно для обозначения ссылки на переменную и операции взятия адреса переменной. Мы рассмотрим вопрос использования амперсанда в главе 10 «Указатели».

Усложненный вариант передачи по ссылке

Здесь мы приведем пример, демонстрирующий менее очевидное применение механизма передачи аргументов по ссылке. Представьте ситуацию, когда вашей программе необходимо работать с парами чисел, упорядоченными по возраста-

нию. Для этого вам необходимо создать функцию, которая сравнивала бы между собой значения двух переменных, и, если первое оказывается больше второго, меняла бы их местами. Приведем листинг программы REFORDER:

```
// reforder.cpp
// упорядочивает по возрастанию пары чисел
#include <iostream>
using namespace std;
int main()
{
  void order( int&, int& );           // прототип
  int n1=99, n2=11;                   // неупорядоченная пара
  int n3=22, n4=88;                   // упорядоченная пара
  order(n1,n2);                       // упорядочивание
  order(n3,n4);                       // обеих пар
  cout << "n1=" << n1 << endl;        // вывод результатов
  cout << "n2=" << n2 << endl;
  cout << "n3=" << n3 << endl;
  cout << "n4=" << n4 << endl;
  return 0;
}
//--------------------------------------------------------
void order( int& numb1, int& numb2 ) // упорядочивает два числа
{
  if( numb1 > numb2 )                 // если первое число больше второго,
  {
    int temp = numb1;                 // то меняем их местами
    numb1 = numb2;
    numb2 = temp;
  }
}
```

В функции main() две пары чисел, причем первая из них не упорядочена, а вторая — упорядочена. Функция order() обрабатывает каждую из этих пар, после чего результат выводится на экран. По результату работы программы мы видим, что значения первой пары поменялись местами, в то время как значения второй пары остались на своих местах:

```
n1=11
n2=99
n3=22
n4=88
```

Параметры функции order() имеют имена numb1 и numb2. В случае, если значение numb1 оказывается меньше numb2, значение numb1 копируется в переменную temp, на его место записывается значение numb2, и затем значение переменной temp копируется в numb2. Обратите внимание на то, что имена numb1 и numb2 указывают на физически различные переменные, в то время как аргументы функции могут быть и одинаковыми. При первом вызове функции order() упорядочивается пара значений n1 и n2, а при втором вызове — пара значений n2 и n3. Таким образом, действие программы заключается в проверке упорядоченности исходных значений и, при необходимости, их сортировке.

Ссылочный механизм напоминает устройство дистанционного управления: вызывающая программа указывает функции переменные, которые нужно обработать, а функция обрабатывает эти переменные, даже не зная их настоящих имен. Это выглядит примерно так, как если бы вы наняли работников, чтобы сделать ремонт в своей комнате, и они, не приходя к вам домой, красили, белили и выполняли другую необходимую работу.

Передача структурных переменных по ссылке

Вы можете применять механизм передачи по ссылке не только к переменным стандартных типов, но и к структурным переменным. Следующая программа, REFERST, производит масштабирование значений типа Distance. Масштабирование представляет собой умножение нескольких длин на один и тот же коэффициент. Так, если исходная длина равна 6'-8", а коэффициент масштабирования равен 0.5, то длина, получаемая в результате масштабирования, равна 3'-4". Масштабирование можно применить, например, к трем измерениям помещения, что приведет к изменению размеров этого помещения с одновременным сохранением его пропорций.

```
// referst.cpp
// передача структурной переменной по ссылке
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
struct Distance                    // длина в английской системе
{
  int feet;
  float inches;
};
///////////////////////////////////////////////////////////
void scale( Distance&, float );    // прототипы функций
void engldisp( Distance );
int main()
{
  Distance d1 = { 12, 6.5 };       //инициализация d1 и d2
  Distance d2 = { 10, 5.5 };
  cout << "d1 = "; engldisp(d1); //вывод исходных значений d1 и d2
  cout << "\nd2 = "; engldisp(d2);
  scale(d1, 0.5);                  // масштабирование d1 и d2
  scale(d2, 0.25);
  cout << "\nd1 = "; engldisp(d1); // вывод новых значений d1 и d2
  cout << "\nd2 = "; engldisp(d2);
  cout << endl;
  return 0;
}
//---------------------------------------------------------
// функция scale()
// масштабирование значения типа Distance
void scale( Distance& dd, float factor )
{
  float inches = (dd.feet*12 + dd.inches)*factor;
  dd.feet = static_cast<int>(inches/12);
  dd.inches = inches - dd.feet *12;
}
```

```
//--------------------------------------------------------
// функция engldisp()
// отображает значение типа Distance на экране
void engldisp( Distance dd )        // параметр dd типа Distance
{
  cout << dd.feet << "\'-" << dd.inches <<"\"";
}
```

Программа инициализирует две переменные типа Distance, d1 и d2, и выводит их значения на экран. Затем вызывается функция scale(), которая производит умножение длины d1 на коэффициент 0.5, а длины d2 — на коэффициент 0.25. После этого на экран выводятся новые значения d1 и d2:

```
d1 = 12'-6.5"
d2 = 10'-5.5"
d1 = 16'-3.25"
d2 = 2'-7.375"
```

Вызовы функции scale() в программе выглядят следующим образом:

```
scale(d1, 0.5);
scale(d2, 0.25);
```

Первый вызов производит масштабирование с коэффициентом 0.5, а второй — с коэффициентом 0.25. Обратите внимание на то, что значения d1 и d2, передаваемые по ссылке, изменяются непосредственно в функции; никакого значения функция при этом не возвращает. Поскольку изменяется только один аргумент, можно было бы модифицировать функцию scale(), передавая ей этот аргумент по значению и возвращая масштабируемую величину с помощью оператора return. В этом случае вызов функции выглядел бы так:

```
d1 = scale(d1, 0.5)
```

• Конечно, такой способ является чересчур длинным.

Замечание о ссылках

В языке С не существует понятия *ссылка*. Схожие с ссылками возможности в С обеспечивают указатели, хотя зачастую их применение менее удобно. Создание ссылочного механизма в С++ было обусловлено стремлением обеспечить гибкость языка в ситуациях, связанных с использованием как объектов, так и простых переменных.

Третьим способом передачи аргументов в функцию является использование указателей. Мы рассмотрим этот способ в главе 10 «Указатели».

Перегруженные функции

Перегруженная функция выполняет различные действия, зависящие от типов данных, передаваемых ей в качестве аргументов. Перегрузка напоминает термос из известного анекдота: один ученый утверждал, что термос — это величайшее изо-

бретение человечества за всю его историю. Когда его спрашивали, почему он так считает, он отвечал: «Эта загадочная вещь позволяет горячему сохранять тепло, а холодному — холод. Как ей удается отличать одно от другого?»

Действительно, кажется загадочным, каким образом функция распознает, какие из действий необходимо совершить над теми или иными данными. Для того чтобы понять суть этого механизма, разберем несколько примеров.

Переменное число аргументов функции

Вспомним функцию starline() из программы TABLE и функцию repchar() из программы TABLEARG, которые мы создали в начале этой главы. Функция starline() выводила на экран линию из 45 символов *, а функция repchar() выводила заданный символ заданное число раз, используя два аргумента. Мы можем создать третью функцию, charline(), которая всегда печатает 45 символов, но позволяет задавать печатаемый символ. Все три функции, starline(), repchar() и charline(), выполняют схожие действия, но их имена различны. Для программиста это означает, что нужно запомнить эти имена и различия между действиями функций. Очевидно, что было бы гораздо удобнее использовать одно и то же имя для всех трех функций, несмотря на то, что они используют различные наборы аргументов. Программа OVERLOAD демонстрирует, каким образом это можно осуществить:

```
// overload.cpp
// перегрузка функций
#include <iostream>
using namespace std;
void repchar();                 // прототипы
void repchar(char);
void repchar(char, int);
int main()
{
  repchar();
  repchar('=');
  repchar('+', 30);
  return 0;
}
//--------------------------------------------------------
// функция repchar()
// выводит на экран 45 символов '*'
void repchar()
{
  for(int j=0; j<45; j++)     // цикл, выполняющийся 45 раз
  cout << '*';                // вывод символа '*'
  cout << endl;
}
//--------------------------------------------------------
// функция repchar()
// выводит 45 заданных символов
void repchar(char ch)
{
  for(int j=0; j<45; j++)     // цикл, выполняющийся 45 раз
  cout << ch;                 // вывод заданного символа
  cout << endl;
}
```

```
//--------------------------------------------------------------
// функция repchar()
// выводит заданный символ заданное число раз
void repchar(char ch, int n)
{
  for(int j=0; j<n; j++)      // цикл, выполняющийся n раз
  cout << ch;                 // вывод заданного символа
  cout << endl;
}
```

Эта программа выводит на экран три различных вида линий:

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
```

Первые две из линий имеют длину 45 символов, третья — 30 символов.

В программе содержатся три функции с одинаковым именем. Каждой из функций соответствует свое объявление, определение и вызов. Каким же образом компилятор различает между собой эти три функции? Здесь на помощь приходит сигнатура функции, которая позволяет различать между собой функции по количеству аргументов и их типам. Другими словами, объявление

```
void repchar();
```

не содержащее аргументов, и объявление

```
void repchar(char, int);
```

задают разные функции.

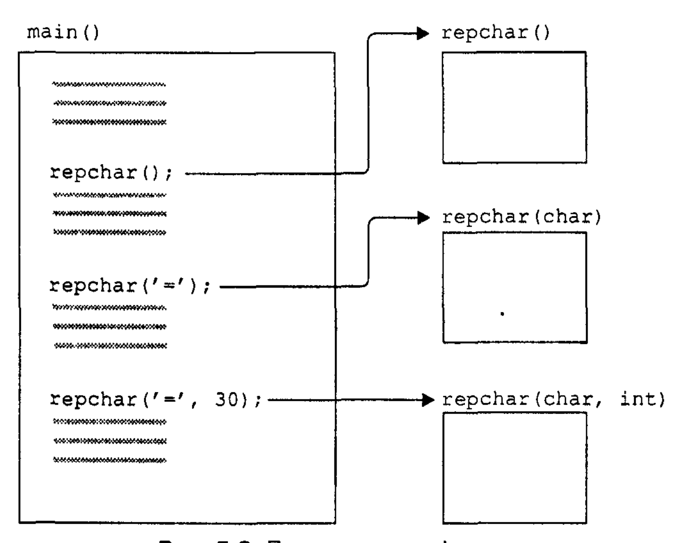


Рис. 5.8. Перегруженные функции

Разумеется, компилятор, обнаружив несколько функций с одинаковыми именами, но разными аргументами, мог бы расценить это как ошибку программиста. Именно таким образом поступил бы компилятор языка C. Но компилятор C++

без излишних вопросов обработает функции для каждого из таких определений. Какая из функций будет выполнена при вызове, зависит от количества аргументов, указанных в вызове. Рисунок 5.8 иллюстрирует данный механизм.

Различные типы аргументов

В примере OVERLOAD мы создали несколько функций, имеющих одно и то же имя, но различное число аргументов. Аналогичным образом можно определить несколько функций с одинаковыми именами и одинаковым количеством аргументов, но различными типами этих аргументов. Следующая программа, OVERENGL, использует перегруженную функцию для вывода расстояния в виде числа футов и числа дюймов. Аргументом функции может являться как структурная переменная типа Distance, так и обычная переменная типа float. В зависимости от типа аргумента, указанного при вызове, будет исполняться одна из двух функций.

```
// overengl.cpp
// демонстрирует перегруженные функции
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
struct Distance             // длина в английской системе
{
  int feet;
  float inches;
};
///////////////////////////////////////////////////////////
void engldisp(Distance);    // прототипы
void engldisp(float);
int main()
{
  Distance d1;              // длина типа Distance
  float d2;                 // длина типа float
  // ввод значения d1
  cout << "\nВведите число футов: "; cin >> d1.feet;
  cout << "Введите число дюймов: "; cin >> d1.inches;
  // ввод значения d2
  cout << "Введите длину в дюймах: "; cin >> d2;
  cout << "\nd1 = ";
  engldisp(d1);             // вывод значения d1
  cout << "\nd2 = ";
  engldisp(d2);             // вывод значения d2
  cout << endl;
  return 0;
}
//--------------------------------------------------------
// функция engldisp()
//
void engldisp(Distance dd )  // параметр dd типа Distance
{
cout <<dd.feet <<"\'-" <<<dd.inches <<"\"";
}
//--------------------------------------------------------------
//engldisp()
// вывод переменной типа float в футах и дюймах
```

```
void engldisp(float dd )    // параметр dd типа float
{
int feet =static_cast<int>(dd /12);
float inches =dd -feet*12;
cout <<feet <<"\'-" <<<inches <<"\"";
}
```

Пользователю предлагается ввести два расстояния: первое — в виде отдельно вводимых количества футов и количества дюймов, а второе — в виде количества дюймов (например, 109,5 дюймов вместо 9'-1,5"). Программа вызывает перегруженную функцию engldisp() для того, чтобы вывести первое значение, имеющее тип Distance(), и второе значение, имеющее тип float. Результат работы программы выглядит следующим образом:

```
Введите число футов: 5
Введите число дюймов: 10.5
Введите расстояние в дюймах: 76.5
d1 = 5'-10.5"
d2 = 6'-4.5"
```

Обратите внимание на то, что, несмотря на одинаковый результат работы различных версий функции engldisp(), код самих функций различен. Той из функций, которая использует в качестве аргумента расстояние в дюймах, необходимо выполнить преобразование вещественного значения к типу Distance() перед тем, как вывести результат на экран.

Использование перегруженных функций упрощает процесс разработки программы, так как у программиста нет необходимости запоминать множество имен функций. Примером того, что в отсутствие перегрузки программа становится очень сложной, может служить вычисление абсолютной величины числа с помощью библиотечных функций C++. Поскольку эти библиотечные функции предназначены не только для C++, но и для C, не поддерживающего перегрузку, для каждого типа данных должна существовать своя функция, вычисляющая модуль. Так, для величин типа int существует функция abs(), для величин комплексного типа — функция cabs(), для величин типа double — функция fabs() и для величин типа long — функция labs(). С помощью перегрузки в C++ единственная функция с именем abs() обрабатывает значения всех указанных типов.

Как мы увидим позже, механизм перегрузки функций полезен при обработке различных типов объектов.

Рекурсия

Существование функций делает возможным использование такого средства программирования, как рекурсия. Рекурсия позволяет функции вызывать саму себя на выполнение. Это может показаться несколько неправдоподобным, и на самом деле зачастую обращение функции к самой себе вызывает ошибку, но при правильном использовании механизм рекурсии может оказаться очень мощным инструментом в руках программиста.

Понять суть рекурсии проще из конкретного примера, чем из долгих пространных объяснений, поэтому обратимся к программе FACTOR, которую мы создали в главе 3 «Циклы и ветвления». В этой программе использовался цикл for для подсчета факториала числа (если вы не помните, что представляет собой факториал числа, прочитайте пояснения к примеру). Наша следующая программа, FACTOR2, вместо цикла for использует рекурсию.

```
// factor2.cpp
// подсчет факториала числа с помощью рекурсии
#include <iostream>
using namespace std;
unsigned long factfunc(unsigned long); // прототип
int main()
{
  int n;                          // число, вводимое пользователем
  unsigned long fact;             // факториал этого числа
  cout << "Введите целое число :";
  cin >> n;
  fact = factfunc(n);
  cout << "Факториал числа " << n << "равен " << fact << endl;
  return 0;
}
//--------------------------------------------------------
// функция factfunc()
// рекурсивно подсчитывает факториал числа
unsigned long factfunc(unsigned long n)
{
  if(n > 1)
    return n * factfunc(n-1);     // вызов самой себя
  else
    return 1;
}
```

Результат работы этой программы такой же, как и у программы FACTOR из главы 3.

Функция main() программы FACTOR2 не содержит ничего необычного: в ней производится вызов функции factfunc(), где в качестве аргумента используется значение, введенное пользователем. Функция factfunc() возвращает функции main() вычисленное значение факториала.

Совсем по-другому обстоит дело с содержимым функции factfunc(). В случае, если параметр n оказывается больше 1, происходит вызов функцией factfunc() самой себя, при этом используется значение аргумента на единицу меньшее, чем текущее значение параметра. Если функция main() вызвала функцию factfunc() с аргументом, равным 5, то сначала функция factfunc() вызовет себя с аргументом, равным 4, затем этот вызов обратится к функции factfunc() с аргументом, равным 3, и т. д. Обратите внимание на то, что каждый экземпляр функции хранит свое значение параметра n во время выполнения рекурсивного вызова.

После того как функция factfunc() вызовет себя четырежды, пятый вызов будет производиться с аргументом, равным 1. Функция узнает об этом с помощью оператора if и, вместо рекурсивного вызова, возвратит четвертому вызову значение, равное 1. Четвертый вызов хранит значение параметра, равное 2, поэтому, произведя умножение параметра на значение, возвращенное пятым вызовом, он получит число 2, которое будет возвращено третьему вызову. Третий вызов хра-

нит значение параметра, равное 3, поэтому второму вызову будет возвращено значение 3*2 = 6. Второй вызов хранит значение параметра, равное 4, поэтому, умножив 4 на 6, возвратит первому вызову значение, равное 24. Наконец, первый вызов умножит 24 на значение параметра, равное 5, и вернет окончательный результат, равный 120, в функцию main().

Таким образом, в этом примере мы имеем пять рекурсивных вызовов с пятью возвращаемыми значениями. Сведем процесс выполнения рекурсии в таблицу:

Версия	Действие	Аргумент или возвращаемое значение
1	Вызов	5
2	Вызов	4
3	Вызов	3
4	Вызов	2
5	Вызов	1
5	Возврат значения	1
4	Возврат значения	2
3	Возврат значения	6
2	Возврат значения	24
1	Возврат значения	120

Каждая рекурсивная функция должна включать в себя условие окончания рекурсии. В противном случае рекурсия будет происходить бесконечно, что приведет к аварийному завершению программы. Ветвление if в функции factfunc() играет роль условия, прекращающего рекурсию, как только параметр достигнет значения, равного 1.

Будут ли все рекурсивные вызовы функции храниться в памяти? Это не совсем так. В памяти будут находиться значения параметров для каждого из вызовов, однако тело функции будет присутствовать в памяти в единственном экземпляре. Но, даже несмотря на это, программа, использующая рекурсии с большим числом вложенных вызовов, может исчерпать все ресурсы оперативной памяти, что повлечет за собой сбой в работе операционной системы.

Встраиваемые функции

Мы уже упоминали, что использование функций является экономичным с точки зрения использования памяти, поскольку вместо дублирования кода используется механизм вызовов функций. Когда компилятор встречает вызов функции, он генерирует команду перехода в эту функцию. После выполнения функции осуществляется переход на оператор, следующий за вызовом функции, как было показано на рис. 5.1.

Использование функций, наряду с сокращением размера памяти, занимаемой кодом, увеличивает время выполнения программы. Для выполнения функции должны быть сгенерированы команды переходов (как правило, это инструкция языка ассемблера CALL или аналогичная ей команда), команды, сохраняющие

значения регистров процессора, команды, помещающие в стек и извлекающие из стека аргументы функции (если они есть), команды, восстанавливающие значения регистров после выполнения функции, и наконец, команда перехода из функции обратно в программу. Кроме того, если функция возвращает значение, то необходимы дополнительные команды, работающие с этим значением. Выполнение всех перечисленных инструкций замедляет работу программы.

Для того чтобы сократить время выполнения небольших функций, вы можете дать указание компилятору, чтобы при каждом вызове такой функции вместо команды перехода производилась подстановка операторов, выполняемых функцией, в код программы. Различия между обычными и встраиваемыми функциями показаны на рис. 5.9.

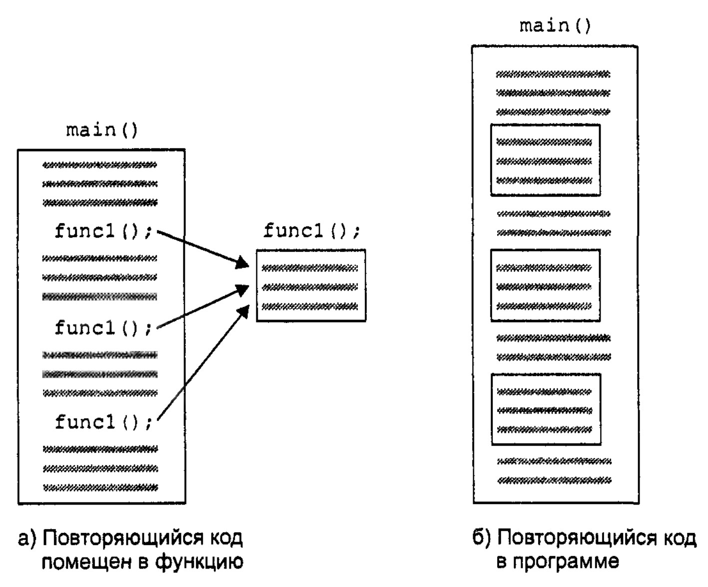


Рис. 5.9. Обычные функции против встраиваемых

Длинные повторяющиеся последовательности действий лучше объединять в обычные функции, поскольку экономия памяти в этом случае оправдывает дополнительные накладные расходы на время выполнения программы. Но если вынести в функцию небольшой фрагмент кода, то выигрыш от экономии памяти будет мал, тогда как дополнительные временные затраты останутся почти на том же уровне, что и для функции большого объема. На самом деле при небольшом размере функции дополнительные инструкции, необходимые для ее вызова, могут занять столько же памяти, сколько занимает код самой функции, поэтому экономия памяти может в конечном счете вылиться в дополнительный ее расход.

В подобных ситуациях вы могли бы вставлять повторяющиеся последовательности операторов в те места программы, где это необходимо, однако в этом случае вы теряете преимущества процедурной структуры программы — четкость и ясность программного кода. Возможно, программа станет работать быстрее и занимать меньше места, но ее код станет неудобным для восприятия.

Решением данной проблемы служит использование встраиваемых функций. Встраиваемые функции пишутся так же, как и обычные, но при компиляции их исполняемый код вставляется, или встраивается, в исполняемый код программы. Листинг программы сохраняет свою организованность и четкость, поскольку функция остается независимой частью программы. В то же время компиляция обеспечивает эффект встраивания кода функции в код программы.

Встраиваемыми следует делать только очень короткие функции, содержащие один-два оператора. Мы приведем небольшую модификацию программы CONVERT2 под названием INLINE, в которой функция lbstokg() сделана встраиваемой.

```
// inline.cpp
// применение встроенных функций
#include <iostream>
using namespace std:
// функция lbstokg()
// переводит фунты в килограммы
inline float lbstokg(float pounds)
{
  return 0.453592*pounds;
}
//-----------------------------------------------------
int main()
{
  float lbs:
  cout << "\nВведите вес в фунтах: ";
  cin >> lbs;
  cout << " " << lbstokg(lbs)
  << endl;
  return 0:
}
```

Для того чтобы сделать функцию встраиваемой, необходимо лишь указать ключевое слово inline в прототипе функции:

```
inline float lbstokg(float pounds)
```

Следует отметить, что ключевое слово inline является лишь рекомендацией компилятору, которая может быть проигнорирована. В этом случае функция будет скомпилирована как обычная. Такое может произойти, например, в том случае, если компилятор посчитает функцию слишком длинной для того, чтобы делать ее встраиваемой.

Если вы знакомы с языком C, то вы отметите, что встраиваемые функции являются аналогом широко используемого в языке C макроса #define. Преимуществом встраиваемых функций по сравнению с макросами является их более корректная работа с типами данных, а также удобный синтаксис.

Аргументы по умолчанию

Можно организовать функцию, имеющую аргументы, таким образом, что ее можно будет вызывать, вообще не указывая при этом никаких аргументов. Однако для этого при объявлении функции необходимо задать значения аргументов по умолчанию.

Чтобы продемонстрировать использование аргументов, задаваемых по умолчанию, приведем пример, являющийся модификацией программы OVERLOAD. В программе OVERLOAD мы использовали три различные функции с одним именем, чтобы обрабатывать различные типы передаваемых значений. Следующая программа, MISARG, достигает того же эффекта, но другим путем:

```
// missarg.cpp
// применение аргументов по умолчанию
#include <iostream>
using namespace std;
void repchar(char='*', int=45); // прототип с аргументами
                                // по умолчанию
int main()
{
  repchar();                    // печатает символ '*' 45 раз
  repchar('=');                 // печатает символ '=' 45 раз
  repchar('+', 30);             // печатает символ '+' 30 раз
  return 0;
}
//--------------------------------------------------------
// функция repchar()
// печатает строку символов
void repchar(char ch, int n)    // при необходимости будут
{                               // использоваться значения по умолчанию
  for(int j=0; j<n; j++)        // цикл n раз
  cout << ch;                   // вывод ch
  cout << endl;
}
```

В этой программе функция repchar() имеет два аргумента. Ее вызов производится из функции main() три раза: первый раз — без аргументов, второй раз — с одним аргументом, третий раз — с двумя аргументами. Первые два вызова работают потому, что вызываемая функция имеет значения аргументов по умолчанию, которые используются в том случае, если при вызове значение аргумента не передается. Значения аргументов по умолчанию определяются в прототипе функции repchar():

```
void repchar(char='*', int=45);   // прототип
```

Значение, присваиваемое аргументу по умолчанию, находится справа от знака равенства, располагающегося после типа аргумента. Подобным образом можно определять значения по умолчанию и в том случае, если в прототипе фигурируют имена переменных:

```
void repchar(char reptChar='*', int numberReps=45);
```

Если при вызове функции число переданных аргументов на единицу меньше, чем описано в прототипе, то компилятор считает, что отсутствует последний аргумент. Таким образом, в случае вызова функции repchar() с одним аргументом значение этого аргумента присваивается параметру ch, а параметру n присваивается значение по умолчанию, равное 45.

Если при вызове repchar() отсутствуют оба аргумента, то обоим ее параметрам, ch и n, будут присвоены значения по умолчанию, соответственно равные '*'

и 45. Таким образом, все три вызова функции repchar() будут корректными, несмотря на то, что количество аргументов в них различно.

Обратите внимание, что опускать при вызове можно только аргументы, стоящие в конце списка при объявлении функции. Например, можно не указать три последних аргумента, но нельзя одновременно пропустить предпоследний аргумент и указать последний. Это вполне логично, поскольку компилятор не может узнать, какой из аргументов, стоящих в середине списка, вы пропустили (можно было бы распознавать отсутствие аргумента с помощью запятых, между которых не указывалось бы отсутствующего значения, но запятые слишком часто приводят к опечаткам, что послужило причиной отказа от этой идеи разработчиков C++). Разумеется, на попытку пропустить те аргументы, для которых не определены значения по умолчанию, компилятор отреагирует выдачей сообщения об ошибке.

Задание значений аргументов по умолчанию полезно в тех случаях, когда аргументы функции часто принимают какое-то одно значение. Кроме того, значения по умолчанию могут использоваться, если программист хочет модифицировать уже написанную функцию путем добавления в нее нового аргумента. В этом случае не нужно будет изменять вызовы функции, поскольку значение нового аргумента будет задано по умолчанию.

Область видимости и класс памяти

Изучив основы работы с функциями, мы рассмотрим два аспекта, касающихся взаимодействия переменных и функций: *область видимости* и *класс памяти*. *Область видимости* определяет, из каких частей программы возможен доступ к переменной, а *класс памяти* — время, в течение которого переменная существует в памяти компьютера. Сначала мы рассмотрим эти понятия вкратце, а затем изучим их более детально.

Рассмотрим два типа области видимости: *локальная область видимости* и *область видимости файла* (еще один тип — *область видимости класса* — мы добавим позже).

Переменные, имеющие локальную область видимости, доступны *внутри того блока, в котором они определены.*

Переменные, имеющие область видимости файла, доступны *из любого места файла, в котором они определены.*

Блоком обычно считается код, заключенный в фигурные скобки. Например, тело функции представляет собой блок.

Существует два класса памяти: automatic (автоматический) и static (статический).

У переменных, имеющих класс памяти automatic, время жизни равно времени жизни функции, внутри которой они определены.

У переменных, имеющих класс памяти static, время жизни равно времени жизни всей программы.

Давайте теперь подробнее рассмотрим, что означают эти понятия.

Локальные переменные

До сих пор в наших примерах мы сталкивались только с переменными, которые определялись внутри тех функций, в которых они использовались. Другими словами, определение переменных происходило внутри фигурных скобок, ограничивающих тело функции:

```
void somefunc()
{
  int somevar;              // переменные, определенные внутри
  float othervar;           // тела функции
  // другие операторы
}
```

Переменные, определяемые внутри функции main(), ничем не отличаются от переменных, определяемых внутри других функций. Переменные, определяемые внутри функции, называют *локальными*, поскольку их область видимости ограничивается этой функцией. Иногда такие переменные называют автоматическими, поскольку они имеют класс памяти automatic.

Рассмотрим область видимости и классы памяти локальных переменных.

Классы памяти локальных переменных

Локальная переменная не существует в памяти до тех пор, пока не будет вызвана функция, в которой она определена (в более общем виде это утверждение звучит так: переменная, определенная внутри любого блока операторов, не существует в памяти до тех пор, пока этот блок не выполнен. Переменные, определенные в теле цикла, существуют только во время исполнения цикла). Так, в приведенном выше фрагменте программы переменные somevar и othervar не существуют до тех пор, пока не произведен вызов функции somefunc(). Это означает, что в памяти нет места, выделенного для их хранения: переменные не определены. Когда управление передается в функцию somefunc(), переменные создаются и под них отводится место в памяти. Затем, когда выполнение функции somefunc() завершается и управление передается вызывающей программе, переменные уничтожаются, а их значения теряются. Название автоматического класса памяти как раз указывает на то, что переменные автоматически создаются при входе в функцию и автоматически уничтожаются при выходе из нее. Период времени между созданием переменной и ее уничтожением называется временем жизни переменной. Время жизни локальной переменной равно времени, проходящему с момента объявления переменной в теле функции до завершения исполнения функции.

Ограниченное время жизни переменной позволяет экономить память во время выполнения программы. В случае, если функция не вызывается, нет необходимости резервировать память под ее локальные переменные. Удаление локальных переменных из памяти при завершении функции позволяет распределить освободившуюся память между локальными переменными других функций.

Область видимости локальных переменных

Область видимости переменной определяет участки программы, из которых возможен доступ к этой переменной. Это означает, что внутри области видимости

переменной операторы могут обращаться к ней по имени и использовать ее значение при вычислении выражений. За пределами области видимости попытки обращения к переменной приведут к выдаче сообщения о том, что переменная неизвестна.

Переменные, определенные внутри функции, видимы, или доступны, только внутри функции, в которой они определены. Предположим, что в вашей программе есть две следующие функции:

```
void somefunc()
{
  int somevar;                  // локальные переменные
  float othervar;
  somevar = 10;                 // корректно
  othervar = 11;                // корректно
  nextvar = 12;                 // некорректно: nextvar невидима в somefunc()
}
void otherfunc()
{
  int nextvar;                  // локальная переменная
  somevar = 20;                 // некорректно: somevar и othervar
  othervar = 21;                // невидимы в otherfunc()
  nextvar = 22;                 // корректно
}
```

Переменная nextvar невидима внутри функции somefunc(), а переменные somevar и othervar невидимы внутри функции otherfunc().

Ограничение области видимости переменной обеспечивает организованность программы и ее модульность. Вы можете быть уверены, что переменные одной функции защищены от несанкционированного доступа других функций, поскольку во всех остальных функциях эти переменные невидимы. Это является важной частью структурного программирования, на котором основывается организация процедурных программ. Ограничение области видимости переменных является важным и для объектно-ориентированного программирования.

В случае, если переменная определяется внутри функции, ее область видимости и время жизни ограничены этой функцией: переменная существует только во время исполнения функции и видима только внутри функции. Как мы позже увидим, не для всех переменных область видимости и время жизни совпадают.

Инициализация

Когда создается локальная переменная, компилятор не пытается собственными силами инициализировать ее. В этом случае переменная имеет неопределенное значение, которое может быть как нулевым, так и представлять собой отличное от нуля число. Для того чтобы переменная получила определенное значение, необходимо явно инициализировать ее:

```
int n = 33;
```

Теперь переменная n начнет свое существование со значением, равным 33.

Глобальные переменные

Следующим видом переменных, который мы рассмотрим, будут *глобальные* переменные. В отличие от локальных переменных, определяемых внутри функций, глобальные переменные определяются вне каких-либо функций (как мы позже увидим, они также определяются вне любого из классов). Глобальная переменная видима из всех функций данного файла и, потенциально, из других файлов. Говоря более точно, глобальная переменная видна из всех функций файла, которые определены позже, чем сама глобальная переменная. Как правило, необходимо, чтобы глобальные переменные были видимы из всех функций, поэтому их определения располагают в начале листинга. Иногда глобальные переменные называют *внешними*, поскольку они являются внешними по отношению к любой из функций программы.

Следующая программа под именем EXTERN демонстрирует использование глобальной переменной тремя функциями.

```
// extern.cpp
// демонстрирует глобальные переменные
#include <iostream>
using namespace std;
#include <conio.h>             // для getch()
char ch = 'a';                 // глобальная переменная ch
void getachar();               // прототипы функций
void putachar();
int main()
{
  while( ch != '\r' )          // main()использует значение ch
  {
    getachar();
    putachar();
  }
  cout << endl;
  return 0;
}
//--------------------------------------------------------
void getachar()                // getachar()использует значение ch
{
  ch = getch();
}
//--------------------------------------------------------
void putachar()                // putachar() использует значение ch
{
  cout << ch;
}
```

Одна из функций, getachar(), считывает символы с клавиатуры. Она использует библиотечную функцию getch(), действие которой напоминает функцию getche() с той разницей, что getch() не отображает вводимые символы на экране. Другая функция, putachar(), отображает вводимые символы на экране. Таким образом, вы видите на экране все символы, которые вы ввели, например

```
Я сижу за компьютером и набираю этот текст
```

Особенностью данной программы является то, что переменная ch не принадлежит какой-либо из функций, а определяется в начале файла, перед объявлениями функций. Переменная ch является глобальной, или внешней. Любая функция, определенная позднее, чем переменная ch, имеет к ней доступ; в случае нашей программы, доступ к переменной ch имеют все три функции: main(), getachar() и putachar(). Таким образом, областью видимости переменной ch является весь исходный файл.

Роль глобальных переменных

Глобальные переменные используются в тех случаях, когда их значения должны быть доступны одновременно нескольким функциям. В процедурно-ориентированных программах глобальные переменные содержат наиболее важную информацию, используемую программой. Тем не менее, как мы уже говорили в главе 1, всеобщая доступность глобальных переменных имеет и негативную сторону: они не защищены от несанкционированного и некорректного доступа со стороны функций. В программах, написанных с применением объектно-ориентированного подхода, необходимость в глобальных переменных гораздо меньше.

Инициализация

Если глобальная переменная инициализируется явно, как показано ниже:

```
int exvar = 199;
```

то подобная инициализация происходит в момент загрузки программы. Если явной инициализации не происходит, например, при следующем определении переменной:

```
int exvar;
```

то в момент создания этой переменной ей автоматически будет присвоено значение, равное 0 (в этом состоит различие между локальными и глобальными переменными. Локальные переменные, не будучи инициализированными явно, после своего создания содержат случайное значение).

Область видимости и время жизни

Глобальные переменные имеют статический класс памяти, что означает их существование в течение всего времени выполнения программы. Память под эти переменные выделяется в начале выполнения программы и закрепляется за ними вплоть до завершения программы. Для того чтобы глобальные переменные были статическими, не обязательно указывать в их определении ключевое слово static: статический класс памяти определен для этих переменных по умолчанию.

Как мы говорили, глобальные переменные видимы от места своего объявления до конца того файла, в котором они определены. Если бы переменная ch была определена между функциями main() и getachar(), то она была бы видна из функций getachar() и putachar(), но не была бы видна из функции main().

Статические локальные переменные

Рассмотрим еще один тип переменных, называемых *статическими локальными* переменными. Существуют и статические глобальные переменные, но они используются только в многофайловых программах, рассмотрение которых мы отложим до главы 13.

Статическая локальная переменная имеет такую же область видимости, как и автоматическая: функцию, к которой принадлежит данная переменная. Однако время жизни у статической локальной переменной иное: оно совпадает со временем жизни глобальной переменной, правда, с той разницей, что существование статической локальной переменной начинается при первом вызове функции, к которой она принадлежит. Далее переменная существует на всем протяжении выполнения программы.

Статические локальные переменные используются в тех случаях, когда необходимо сохранить значение переменной в памяти после того, как выполнение функции будет завершено, или, другими словами, между вызовами функций. В следующем примере функция getavg() подсчитывает среднее арифметическое значений, полученных ею, с учетом нового переданного ей значения. Функция запоминает количество переданных ей значений и с каждым новым вызовом увеличивает это значение на единицу. Функция возвращает среднее арифметическое, полученное путем деления суммы всех переданных значений на количество этих значений. Текст программы STATIC приведен в листинге:

```
// static.cpp
// демонстрирует статические локальные переменные
#include <iostream>
using namespace std;
float getavg(float);          // прототип функции
int main()
{
  float data=1, avg;
  while( data != 0 )
  {
    cout << "Введите число: ";
    cin >> data;
    avg = getavg(data);
    cout << "Среднее значение: " << avg << endl;
  }
  return 0;
}
//---------------------------------------------------------
// функция getavg()
// находит среднее арифметическое всех введенных значений
float getavg(float newdata)
{
  static float total = 0;     // инициализация статических
  static int count = 0;       // переменных при первом вызове
  count++;                    // увеличение счетчика
  total += newdata;           // добавление нового значения к сумме
  return total/count;         // возврат нового среднего значения
}
```

Далее приведем пример результата работы программы:

```
Введите число: 10
Среднее значение: 10
Введите число: 20
Среднее значение: 15
Введите число: 30
Среднее значение: 20
```

Статические переменные total и count функции getavg() сохраняют свои значения после завершения этой функции, поэтому при следующем вызове этими значениями снова можно пользоваться.

Инициализация

Инициализация статических переменных, таких, как total и count в функции getavg(), происходит один раз — во время первого вызова функции. При последующих вызовах повторной инициализации не происходит, как это должно было бы произойти с обычными локальными переменными.

Класс памяти

Если вы немного знакомы с тем, как устроена операционная система, то вам, вероятно, будет интересен тот факт, что локальные переменные и аргументы функций хранятся в стеке, а глобальные и статические переменные находятся в динамической области памяти.

Таблица 5.2. Области видимости и классы памяти переменных

	Локальная	Статическая локальная	Глобальная
Области видимости	Функция	Функция	Программа
Время жизни	Функция	Программа	Программа
Начальное значение	Случайное	0	0
Область памяти	Стек	Динамическая	Динамическая
Назначение	Переменные, используемые отдельной функцией, уничтожающиеся при выходе из нее	Переменные, используемые отдельной функцией, но сохраняющие свои значения между вызовами функции	Переменные, используемые несколькими функциями

Возвращение значения по ссылке

Теперь, когда мы рассмотрели глобальные переменные, мы можем перейти к рассмотрению еще одного аспекта программирования на C++, который, возможно, покажется несколько нестандартным. Подобно тому как мы передавали в функцию аргументы с помощью ссылок, мы можем возвращать значение функции по ссылке. Причины для использования ссылочного механизма при возвращении значения функцией пока в основном будут вам не совсем понятны. Одной из причин, как мы увидим в главе 11 «Виртуальные функции», является необходимость избежать копирования объектов большого размера. Другой причиной явля-

ется открывающаяся возможность использовать вызов функции в качестве левого операнда операции присваивания. Чтобы не тратить время на малопонятные словесные манипуляции, приведем пример программы RETREF, иллюстрирующей данный механизм в действии:

```
// retref.cpp
// возвращение значения по ссылке
#include <iostream>
using namespace std;
int x;                      // глобальная переменная
int& setx();                // прототип функции
int main()
{                           // присваивание значения x при
  setx() = 92;              // помощи вызова функции слева
  cout << "x=" << x << endl; // вывод нового значения x
  return 0;
}
//--------------------------------------------------------
int& setx()
{
  return x;                 // возвращает значение, которое будет изменено
}
```

В этой программе функция setx(), согласно своему прототипу, имеет тип возвращаемого значения int&:

```
int& setx();
```

Внутри функции содержится оператор

```
return x;
```

где переменная x была определена как глобальная. Теперь у вас появляется на первый взгляд странная возможность располагать вызов функции слева от знака равенства в операции присваивания:

```
setx() = 92;
```

В результате переменной, возвращаемой функцией, присваивается значение, стоящее справа от знака равенства. Таким образом, оператор, приведенный выше, присваивает переменной x значение, равное 92. Это подтверждается и результатом работы программы на экране:

```
x=92
```

Вызов функции в качестве левого операнда операции присваивания

Вспомним, что вызовы обычных функций, в том случае, если они возвращают значение, могут использоваться как само это значение:

```
y = squareroot(x);
```

Здесь значение, возвращаемое функцией squareroot(), присваивается переменной y. Вызов функции интерпретируется как значение, получаемое при его

выполнении. С другой стороны, функция, возвращающая ссылку, интерпретируется как переменная. Возвращение функцией ссылки равносильно возврату псевдонима переменной, входящей в оператор return. В программе RETREF функция setx() возвращает ссылку на переменную x. Это и объясняет тот факт, что вызов функции setx() может использоваться в качестве левого операнда операции присваивания.

Из вышесказанного следует вывод: с помощью ссылки вы не можете возвратить из функции константу. Нельзя определять функцию setx() следующим образом:

```
int& setx()
{
  return 3:
}
```

Если попытаться скомпилировать программу, содержащую такую функцию, то компилятор сообщит о необходимости наличия lvalue, то есть некоторого элемента программы, которому можно присвоить значение. Таким элементом может являться переменная, но не может являться константа.

Можно попытаться обойти это ограничение следующим образом, введя локальную переменную:

```
int& setx()
{
   int x = 3:
   return x;                    // ошибка
}
```

Однако и в этом случае компилятор возвратит ошибку. Проблема заключается в том, что при выходе из функции локальные переменные уничтожаются, и действие по возвращению ссылки на несуществующую переменную теряет смысл.

Зачем нужно возвращение по ссылке?

Конечно, все вышесказанное не снимает вопроса о том, зачем использовать вызовы функций слева от знака присваивания. На самом деле в практике процедурного программирования существует лишь небольшое число ситуаций, где можно использовать такой прием. Как и в продемонстрированном примере, как правило, гораздо проще добиться желаемого результата с помощью других методов. Тем не менее, в главе 8 «Перегрузка операций» мы столкнемся с ситуацией, требующей применения возвращения значения по ссылке. Пока для вас будет вполне достаточно лишь представления об этом механизме.

Константные аргументы функции

Мы видели, что ссылочный механизм передачи аргументов в функцию позволяет функции изменять значения аргументов. Но существуют и другие причины для использования ссылок на аргументы функции, и одной из таких причин явля-

ется эффективность. Некоторые переменные, используемые в качестве аргументов функции, могут иметь большой размер; примером являются структурные переменные с большим числом полей. В случае, если аргумент занимает много места в памяти, его передача по ссылке является гораздо более эффективной, поскольку в последнем случае в функцию передается не значение переменной, а только ее адрес.

Предположим, что вы используете ссылочный механизм только из соображений эффективности и при этом не хотите, чтобы функция имела свободный доступ к аргументу и изменяла его значение. Для того чтобы получить подобную гарантию, вы можете указать модификатор const перед соответствующим аргументом в прототипе функции. Пример CONSTARG демонстрирует применение этого метода.

```
// constarg.cpp
// демонстрирует константные аргументы функций
void aFunc( int& a, const int& b);   // прототип функции
int main()
{
  int alpha = 7;
  int beta = 11;
  aFunc(alpha, beta);
  return 0;
}
//----------------------------------------------------------
void aFunc(int& a, const int& b)   // определение функции
{
 a = 107;                          // корректно
 b = 111;                          // ошибка при попытке
                                   // изменить константный аргумент
}
```

В данной программе мы хотим быть уверенными в том, что функция aFunc() не сможет изменить значения переменной beta (в то же время нам не нужно подобной уверенности относительно переменной alpha). Мы используем модификатор const с переменной beta в прототипе функции и ее определении:

```
void aFunc(int& alpha, const int& beta);
```

Теперь любая попытка изменить содержимое переменной beta функцией aFunc() повлечет за собой сообщение об ошибке от компилятора. Одним из положений идеологии разработчиков C++ является такое: лучше обнаруживать ошибки в программе на этапе ее компиляции, чем во время ее выполнения. Использование модификатора const демонстрирует этот принцип в действии.

Если вы желаете передать в функцию константную переменную с помощью ссылки, у вас не остается другого выбора, как использовать модификатор const в прототипе функции (с передачей константного аргумента по значению не возникает подобной проблемы; определение константной переменной гарантирует, что ни функция, ни любые операторы программы не способны изменить ее значения).

Многие библиотечные функции используют константные аргументы подобным образом, как мы увидим в следующих наших примерах.

Резюме

Функции позволяют обеспечить внутреннюю организованность программы и сократить размер ее кода, присваивая повторяющимся фрагментам программы имя и заменяя этим именем все повторения, встречающиеся в программе. Объявление, или прототип функции задает общий вид функции, вызов функции передает управление в функцию, а определение функции задает совокупность действий, выполняемых функцией. Первая строка в определении функции называется спецификатором функции.

Передача аргументов в функцию может осуществляться по значению и по ссылке. В первом случае функция работает с копией значения аргумента, во втором функции доступна сама переменная, передаваемая в качестве аргумента.

Функция может возвращать только одно значение. Как правило, функция производит возврат по значению, но возможен и возврат по ссылке, что позволяет использовать вызов функции в качестве левого операнда операции присваивания. Структурные переменные могут выступать как в роли аргументов, так и в роли возвращаемых значений.

Перегруженная функция представляет собой группу функций, имеющих одно и то же имя. Какая из функций выполняется при вызове, зависит от количества указанных в вызове аргументов, а также их типов.

Встроенные функции внешне похожи на обычные функции, но при вызовах их код вставляется непосредственно в исполняемый код программы. Встроенные функции исполняются быстрее, но могут занимать в памяти больше места, чем обычные функции, если только размер встроенных функций не является очень маленьким.

Если в функции используются значения аргументов по умолчанию, то при вызове функции не обязательно указывать значения этих аргументов. Вместо отсутствующих значений будут использоваться значения по умолчанию.

Переменные имеют характеристику, которая называется класс памяти. Наиболее простым и распространенным классом памяти является автоматический. Локальные переменные имеют автоматический класс памяти: они существуют только до тех пор, пока не завершится исполнение вызова функции. Кроме того, эти переменные видны только внутри тела функции. Глобальные переменные имеют статический класс памяти: время их существования определяется временем выполнения программы. Кроме того, глобальные переменные видимы во всем исходном файле, начиная с места их объявления. Статические локальные переменные существуют на всем протяжении процесса выполнения программы, но область их видимости ограничена той функцией, к которой они принадлежат.

Функция не может изменять значений тех аргументов, которые описаны в ее прототипе с модификатором const. Переменная, определенная в вызывающей функции как константная, автоматически защищена от изменения функцией.

В главе 4 мы изучили первую из двух основных составляющих объектов — структуры, представляющие собой объединения данных. В этой главе познакомились со второй составляющей — функциями. Теперь мы готовы объединить эти две компоненты в одну и приступить к созданию объектов, чем мы и займемся в следующей главе.

Вопросы

Ответы на нижеприведенные вопросы можно найти в приложении Ж.

1. Наиболее важным из назначений функции является:
   а) именование блока операторов;
   б) уменьшение размера программы;
   в) обработка аргументов и возвращение значения;
   г) структуризация программы.
2. Код функции задается в ________ функции.
3. Напишите функцию foo(), выводящую на экран слово foo.
4. Оператор, описывающий функцию, называется ее ________ или ________.
5. Операторы, выполняющие назначение функции, составляют ______ функции.
6. Оператор, инициирующий выполнение функции, называется _____ функции.
7. Первая строка в объявлении функции называется __________.
8. Аргумент функции — это:
   а) переменная функции, получающая значение из вызывающей программы;
   б) способ, с помощью которого функция защищает себя от воздействия значений, передаваемых вызывающей программой;
   в) значение, передаваемое вызывающей программой в функцию;
   г) значение, возвращаемое функцией вызывающей программе.
9. Истинно ли следующее утверждение: когда аргументы передаются по значению, функция имеет доступ к переменным вызывающей программы?
10. Для чего предназначена возможность указывать в прототипе функции имена аргументов?
11. Какие из перечисленных ниже элементов программы можно передавать в функцию:
    а) константы;
    б) переменные;
    в) структуры;
    г) заголовочные файлы.
12. Что означают пустые скобки после имени функции?
13. Сколько значений может возвращать функция?
14. Истинно ли следующее утверждение: когда функция возвращает значение, ее вызов можно ставить справа от знака операции присваивания?
15. Где указывается тип значения, возвращаемый функцией?
16. Функция, не возвращающая значения, имеет тип возвращаемого значения __________.

17. Дана следующая функция:

```
int times2(int a)
   {
     return (a*2);
 }
```

Напишите функцию main(), которая будет содержать все необходимое для вызова данной функции.

18. Когда аргумент передается в функцию по ссылке,

а) внутри функции создается переменная, хранящая значение этого аргумента;

б) функция не имеет доступа к значению аргумента;

в) в вызывающей программе создается временная переменная для хранения значения аргумента;

г) функция получает доступ к аргументу в вызывающей программе.

19. Какова причина использования ссылочного механизма передачи аргументов в функцию?

20. Перегруженные функции:

а) являются группой функций, имеющих одно и то же имя;

б) имеют одинаковое количество аргументов и их типы;

в) облегчают процесс программирования;

г) могут не выдержать нагрузки.

21. Напишите прототипы двух перегруженных функций с именем bar(). Обе функции имеют возвращаемое значение типа int. Первая функция имеет единственный аргумент типа char, а вторая — два аргумента типа char. Если это невозможно, объясните причину.

22. Как правило, встроенные функции исполняются ______, чем обычные, но занимают ______ места в памяти.

23. Напишите прототип встроенной функции с именем foobar(), имеющей один аргумент типа float и возвращающей значение типа float.

24. Значение аргумента по умолчанию:

а) может использоваться вызывающейся программой;

б) может использоваться функцией;

в) должно быть константой;

г) должно быть значением переменной.

25. Напишите прототип функции с именем blyth(), возвращающей значение типа char и принимающей два аргумента. Первый из аргументов имеет тип int, а второй — тип float и значение по умолчанию, равное 3.14159.

26. Область видимости и класс памяти связаны с ______ и ______ переменной.

27. Какие функции могут иметь доступ к глобальной переменной, расположенной в одном файле с ними?
28. Какие функции имеют доступ к локальной переменной?
29. Статическая локальная переменная используется для:
    а) расширения области видимости переменной;
    б) ограничения области видимости переменной до одной функции;
    в) сохранения переменной в памяти после выполнения функции;
    г) сохранения значения переменной после завершения функции.
30. В каком необычном месте программы можно использовать вызов функции, если эта функция возвращает значение по ссылке?

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Вернитесь к рассмотрению примера CIRCAREA главы 2 «Основы программирования на C++». Напишите функцию с именем circarea(), которая аналогичным образом вычисляет площадь круга. Функция должна принимать один аргумент типа float и возвращать значение типа float. Напишите функцию main(), которая просит пользователя ввести значение радиуса, вызывает функцию circarea(), а затем отображает результат вычисления на экране.

*2. Возведение числа n в степень p — это умножение числа n на себя p раз. Напишите функцию с именем power(), которая в качестве аргументов принимает значение типа double для n и значение типа int для p и возвращает значение типа double. Для аргумента, соответствующего степени числа, задайте значение по умолчанию, равное 2, чтобы при отсутствии показателя степени при вызове функции число n возводилось в квадрат. Напишите функцию main(), которая запрашивает у пользователя ввод аргументов для функции power(), и отобразите на экране результаты ее работы.

*3. Напишите функцию с именем zeroSmaller(), в которую передаются с помощью ссылок два аргумента типа int, присваивающую меньшему из аргументов нулевое значение. Напишите программу для проверки работы функции.

*4. Напишите функцию, принимающую в качестве аргументов два значения типа Distance и возвращающую значение наибольшего из аргументов (необходимую информацию можно найти в примере RETSTRC).

5. Напишите функцию с именем hms_to_secs(), имеющую три аргумента типа int: часы, минуты и секунды. Функция должна возвращать эквивалент переданного ей временного значения в секундах (типа long). Создайте программу, которая будет циклически запрашивать у пользователя ввод значения часов, минут и секунд и выводить результат работы функции на экран.

6. Модифицируйте программу, описанную в упражнении 11 главы 4 «Структуры», складывающую два структурных значения типа time. Теперь программа должна включать в себя две функции. Первая, time_to_secs(), принимает в качестве аргумента значение типа time и возвращает эквивалентное значение в секундах типа long. Вторая, secs_to_time(), в качестве аргумента принимает число секунд, имеющее тип long, а возвращает эквивалентное значение типа time.
7. Взяв в качестве основы функцию power() из упражнения 2, работающую только со значением типа double, создайте перегруженные функции с этим же именем, принимающими в качестве аргумента значения типа char, int, long и float. Напишите программу, вызывающую функцию power() со всеми возможными типами аргументов.
8. Напишите функцию с именем swap(), обменивающую значениями два своих аргумента типа int (обратите внимание, что изменяться должны значения переменных из вызывающей программы, а не локальных переменных функции). Выберите способ передачи аргументов. Напишите вызывающую программу main(), использующую данную функцию.
9. Переработайте программу из упражнения 8 так, чтобы функция swap() принимала в качестве аргументов значения типа time (см. упражнение 6).
10. Напишите функцию, которая при каждом вызове будет выводить на экран количество раз, которое она вызывалась ранее. Напишите программу, которая будет вызывать данную функцию не менее 10 раз. Попробуйте реализовать данную функцию двумя различными способами: с использованием глобальной переменной и статической локальной переменной для хранения числа вызовов функции. Какой из способов предпочтительней? Почему для решения задачи нельзя использовать обычную локальную переменную?
11. Напишите программу, использующую структуру sterling, которая описана в упражнении 10 главы 4. Программа должна получать от пользователя значения двух денежных сумм, выраженных в фунтах, шиллингах и пенсах, складывать эти значения и выводить результат на экран в том же формате. Необходимо разработать три функции. Первая из них должна получать от пользователя число фунтов, шиллингов и пенсов и возвращать соответствующее значение типа sterling. Вторая функция должна принимать в качестве аргументов два значения типа sterling, складывать их и возвращать значение, также имеющее тип sterling. Третья функция должна принимать аргумент типа sterling и выводить его значение на экран.
12. Модифицируйте калькулятор, созданный в упражнении 12 главы 4, так, чтобы каждая арифметическая операция выполнялась с помощью функции. Функции могут называться fadd(), fsub(), fmul() и fdiv(). Каждая из функций должна принимать два структурных аргумента типа fraction и возвращать значение того же типа.

Глава 6
Объекты и классы

- ✦ Простой класс
- ✦ Объекты C++ и физические объекты

Теперь мы приступим к изучению того раздела программирования, к которому так долго готовились: *объекты и классы*. Мы предварительно рассмотрели все, что нам будет необходимо: структуры, позволяющие группировать данные, и функции, объединяющие фрагменты программы под одним именем. В этой главе мы соединим эти два понятия и создадим новый элемент программы — класс. Мы начнем с создания самых простых классов, постепенно усложняя наши примеры. Вначале нам будет необходимо сосредоточить свое внимание на частностях, касающихся создания классов и объектов, однако в конце главы мы поговорим об общих аспектах объектно-ориентированного подхода к программированию.

В процессе чтения этой главы рекомендуется при необходимости возвращаться к материалу главы 1 «Общие сведения».

Простой класс

Наш первый пример содержит класс и два объекта этого класса. Несмотря на свою простоту, он демонстрирует синтаксис и основные черты классов C++. Листинг программы SMALLOBJ приведен ниже[1].

```
// smallobj.cpp
// демонстрирует простой небольшой объект
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class smallobj                  // определение класса
{
  private:
    int somedata;               // поле класса
  public:
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
    void setdata(int d)      // метод класса, изменяющий значение поля
        { somedata = d; }
    void showdata()          // метод класса, отображающий значение поля
      { cout << "Значение поля равно " << somedata << endl; }
};
//////////////////////////////////////////////////////////
int main()
{
  smallobj s1, s2;           // определение двух объектов класса smallobj
  s1.setdata(1066);          // вызовы метода setdata()
  s2.setdata(1776);
  s1.showdata();             // вызовы метода showdata()
  s2.showdata();
  return 0;
}
```

Класс smallobj, определенный в этой программе, содержит одно поле данных и два метода. Методы обеспечивают доступ к полю данных класса. Первый из методов присваивает полю значение, а второй метод выводит это значение на экран (возможно, незнакомые термины привели вас в недоумение, но скоро мы раскроем их смысл).

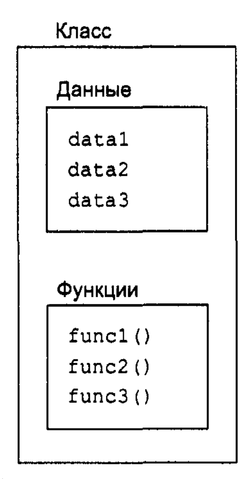


Рис. 6.1. Класс содержит данные и функции

Объединение данных и функций является стержневой идеей объектно-ориентированного программирования. Это проиллюстрировано на рис. 6.1.

Классы и объекты

В главе 1 мы говорили, что объект находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу. Объект является экземпляром класса, так же, как автомобиль является экземпляром колесного средства передвижения. Класс smallobj определяется в начале программы SMALLOBJ. Позже, в функции main(), мы определяем два объекта s1 и s2, являющихся экземплярами класса smallobj.

Каждый из двух объектов имеет свое значение и способен выводить на экран это значение. Результат работы программы выглядит следующим образом:

```
Значение поля равно 1076 - вывод объекта s1
Значение поля равно 1776 - вывод объекта s2
```

Рассмотрим подробнее первую часть программы, где происходит определение класса smallobj. Затем мы обратимся к функции main(), в которой задействованы объекты класса smallobj.

Определение класса

Определение класса smallobj в приведенной выше программе выглядит следующим образом:

```
class smallobj              // определение класса
{
  private:
    int somedata;            // поле класса
  public:
    void setdata(int d)      // метод класса, изменяющий значение поля
      { somedata = d; }
    void showdata()          // метод класса, отображающий значение поля
      { cout << "Значение поля равно " << somedata << endl; }
};
```

Определение начинается с ключевого слова class, за которым следует имя класса; в данном случае этим именем является smallobj. Подобно структуре, тело класса заключено в фигурные скобки, после которых следует точка с запятой (;) (не забывайте ставить этот знак. Конструкции, связанные с типами данных, такие, как структуры и классы, требуют после своего тела наличия точки с запятой, в отличие от конструкций, связанных с передачей управления, например функций и циклов).

private и public

Тело класса содержит два не встречавшихся раньше ключевых слова: private и public. Сейчас мы раскроем их смысл.

Ключевой особенностью объектно-ориентированного программирования является возможность *сокрытия данных*. Этот термин понимается в том смысле, что данные заключены внутри класса и защищены от несанкционированного доступа функций, расположенных вне класса. Если необходимо защитить какие-либо данные, то их помещают внутрь класса с ключевым словом private. Такие данные доступны только внутри класса. Данные, описанные с ключевым словом public, напротив, доступны за пределами класса. Вышесказанное проиллюстрировано на рис. 6.2.

Зачем скрывать данные?

Не путайте сокрытие данных с техническими средствами, предназначенными для защиты баз данных. В последнем случае для обеспечения сохранности дан-

ных можно, например, попросить пользователя ввести пароль перед тем, как разрешить ему доступ к базе данных. Пароль обеспечивает защиту базы данных от несанкционированного или злоумышленного изменения, а также копирования и чтения ее содержимого.

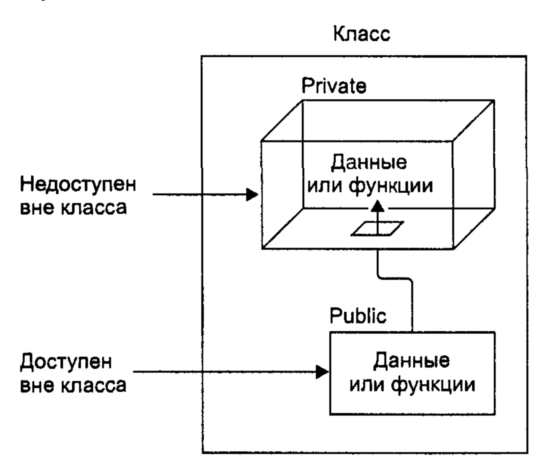


Рис. 6.2. Скрытые и общедоступные классы

Сокрытие данных в нашем толковании означает ограждение данных от тех частей программы, которые не имеют необходимости использовать эти данные. В более узком смысле это означает сокрытие данных одного класса от другого класса. Сокрытие данных позволяет уберечь опытных программистов от своих собственных ошибок. Программисты могут сами создать средства доступа к закрытым данным, что значительно снижает вероятность случайного или некорректного доступа к ним.

Данные класса

Класс smallobj содержит всего одно поле данных somedata, имеющее тип int. Данные, содержащиеся внутри класса, называют *данными-членами* или *полями класса*. Число полей класса, как и у структуры, теоретически может быть любым. Поскольку перед описанием поля somedata стоит ключевое слово private, это поле доступно только внутри класса.

Методы класса

Методы класса — это функции, входящие в состав класса. Класс smallobj содержит два метода: setdata() и showdata(). Тела обоих методов состоят из одного оператора, который записан на одной строке с фигурными скобками, ограничивающими тело функции. Разумеется, можно использовать и более традиционный способ оформления функций:

```
void setdata(int d)
{
  somedata = d;
}
```

и

```
void showdata()
{
  cout << "\nЗначение поля равно " << somedata;
}
```

В тех случаях, когда тела методов невелики по размеру, имеет смысл использовать более сжатую форму их записи.

Поскольку методы setdata() и showdata() описаны с ключевым словом public, они доступны за пределами класса smallobj. Мы покажем, каким образом можно получить доступ к этим функциям, чуть позже. На рис. 6.3 показан синтаксис определения класса.

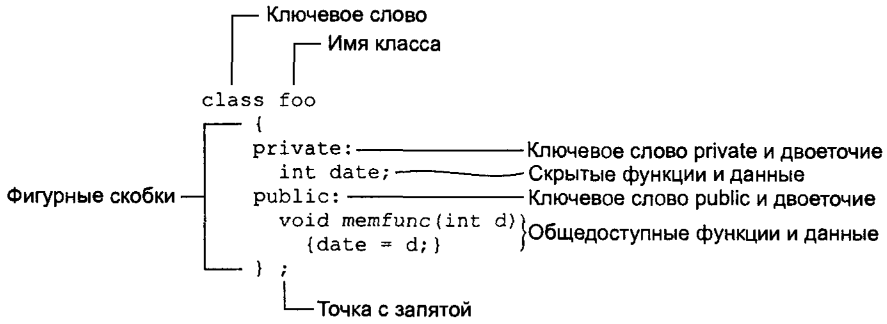


Рис. 6.3. Синтаксис определения класса

Сокрытие данных и доступность функций

Как правило, скрывая данные класса, его методы оставляют доступными. Это объясняется тем, что данные скрывают с целью избежать нежелательного внешнего воздействия на них, а функции, работающие с этими данными, должны обеспечивать взаимодействие между данными и внешней по отношению к классу частью программы. Тем не менее, не существует четкого правила, которое бы определяло, какие данные следует определять как private, а какие функции — как public. Вы можете столкнуться с ситуациями, когда вам будет необходимо скрывать функции и обеспечивать свободный доступ к данным класса.

Методы класса внутри определения класса

Методы класса smallobj выполняют действия, типичные для методов классов вообще: они считывают и присваивают значения полям класса. Метод setdata() принимает аргумент и присваивает полю somedata значение, равное значению аргумента. Метод showdata() отображает на экране значение поля somedata.

Обратите внимание на то, что функции setdata() и showdata() определены внутри класса, то есть код функции содержится непосредственно в определении класса (здесь определение функции не означает, что код функции помещается в память. Это происходит лишь при создании объекта класса). Методы класса, определенные подобным образом, по умолчанию являются встраиваемыми (встраиваемые функции обсуждались в главе 5 «Функции»). Позже мы увидим,

что функции внутри класса можно не только определять, но и объявлять, а определение функции производить в другом месте. Функция, определенная вне класса, по умолчанию уже не является встраиваемой.

Использование класса

Теперь, когда класс определен, давайте рассмотрим, каким образом его можно использовать в функции main(). Мы увидим, как определяются объекты и каким образом организован доступ к методам уже определенных объектов.

Определение объектов

Первым оператором функции main() является

```
smallobj s1, s2;
```

Этот оператор определяет два объекта s1 и s2 класса smallobj. Обратите внимание на то, что при определении класса smallobj не создаются никакие его объекты. Определение класса лишь задает вид будущего объекта, подобно тому, как определение структуры не выделяет память под структурные переменные, а лишь описывает их организацию. Все операции программа производит с объектами. Определение объекта похоже на определение переменной: оно означает выделение памяти, необходимой для хранения объекта.

Вызов методов класса

Следующая пара операторов осуществляет вызов метода setdata():

```
s1.setdata(1066);
s1.setdata(1776);
```

Эти операторы выглядят не так, как обычный вызов функции. Почему имена объектов s1 и s2 связаны с именами функций операцией точки (.)? Такой странный синтаксис объясняется тем, что вызов применим к методу конкретного объекта. Поскольку setdata() является методом класса smallobj, его вызов должен быть связан с объектом этого класса. Например, оператор

```
setdata(1066);
```

сам по себе не имеет смысла, потому что метод всегда производит действия с конкретным объектом, а не с классом в целом. Попытка доступа к классу по смыслу сходна попытке сесть за руль чертежа автомобиля. Кроме бессмысленности такого действия, компилятор расценил бы это как ошибку. Таким образом, доступ к методам класса возможен только через конкретный объект этого класса.

Для того чтобы получить доступ к методу класса, необходимо использовать операцию точки (.), связывающую метод с именем объекта. Синтаксически это напоминает доступ к полям структуры, но скобки позади имени метода говорят о том, что мы совершаем вызов функции, а не используем значение переменной (операцию точки называют *операцией доступа к члену класса*).

Оператор

```
s1.setdata(1066);
```

вызывает метод setdata() объекта s1. Метод присваивает полю somedata объекта s1 значение, равное 1066. Вызов

```
s2.setdata(1776);
```

подобным же образом присваивает полю somedata объекта s2 значение, равное 1776. Теперь мы имеем два объекта с различными значениями поля somedata, как показано на рис. 6.4.

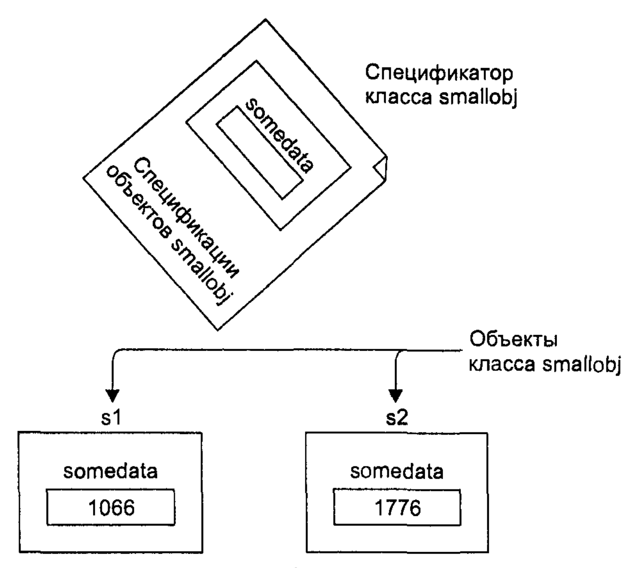


Рис. 6.4. Два объекта класса smallobj

Аналогично, два вызова функции showdata() отобразят на экране значения полей соответствующих объектов:

```
s1.showdata();
s2.showdata();
```

Сообщения

В некоторых объектно-ориентированных языках программирования вызовы методов объектов называют *сообщениями*. Так, например, вызов

```
s1.showdata();
```

можно рассматривать как посылку сообщения объекту s1 с указанием вывести на экран свои данные. Термин *сообщение* не входит в число формальных терминов языка C++, однако его полезно держать в голове в процессе дальнейшего обсуждения. Представление вызова методов в виде сообщений подчеркивает независимость объектов как самостоятельных единиц, взаимодействие с которыми осуществляется путем обращения к их методам. Если обратиться к аналогии

со структурой компании, приведенной в главе 1, то вызов метода будет похож на письмо к секретарю отдела продаж с запросом статистики о рынке компании в каком-либо регионе.

Объекты программы и объекты реального мира

Зачастую объекты, использующиеся в программе, представляют реальные физические объекты. В таких ситуациях проявляется взаимодействие между программой и реальным миром. Мы рассмотрим две подобные ситуации: детали изделия и рисование кругов.

Детали изделия в качестве объектов

Класс smallobj из предыдущего примера содержал только одно поле данных. Теперь мы рассмотрим более интересный пример. Мы создадим класс, основой для которого послужит структура, описывающая комплектующие изделия и ранее использовавшаяся в программе PARTS главы 4 «Структуры». Рассмотрим следующий пример — OBJPART:

```
// objpart.cpp
// детали изделия в качестве объектов
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class part                      // определение класса
{
  private:
  int modelnumber;              // номер изделия
  int partnumber;               // номер детали
  float cost;                   // стоимость детали
  public:
    // установка данных
    void setpart(int mn, int pn, float c)
   {
      modelnumber = mn;
      partnumber = pn;
      cost = c;
    }
    void showpart()             // вывод данных
    {
      cout << "Модель " << modelnumber;
      cout << ", деталь " << partnumber;
      cout << ", стоимость $" << cost << endl;
    }
};
////////////////////////////////////////////////////////////
int main()
{
  part part1;                   // определение объекта
                                // класса part
```

```
  part1.setpart(6244. 373. 217.55F): // вызов метода
  part1.showpart():                  // вызов метода
  return 0:
}
```

В этой программе используется класс part. В отличие от класса smallobj, класс part состоит из трех полей: modelnumber, partnumber и cost. Метод класса setpart() присваивает значения всем трем полям класса одновременно. Другой метод, showpart(), выводит на экран содержимое полей.

В примере создается единственный объект класса part с именем part1. Метод setpart() присваивает его полям значения соответственно 6244, 373 и 217.55. Затем метод showpart() выводит эти значения на экран. Результат работы программы выглядит следующим образом:

```
Модель 6244. деталь 373. цена $217.55
```

Этот пример уже ближе к реальной жизни, чем SMALLOBJ. Если бы вы разрабатывали инвентаризационную программу, то, вероятно, создали бы класс, аналогичный классу part. Мы привели пример объекта C++, моделирующего реально существующий объект — комплектующие изделия.

Круги в качестве объектов

В следующем нашем примере мы создадим объект, представляющий собой круг, отображающийся на вашем экране. Круг не столь материален, как деталь, которую легко можно подержать в руке, но тем не менее вы сможете увидеть его изображение, когда запустите программу.

Наша программа будет представлять собой объектно-ориентированную версию программы CIRCSTRC главы 5 (как и в программе CIRCSTRC, вам будет необходимо включить функции консольной графики в ваш проект. Эти файлы можно загрузить с сайта издательства, адрес которого указан во введении к данной книге. Описание файлов содержится в приложении Д «Упрощенный вариант консольной графики». Кроме того, полезную информацию можно также найти в приложениях к вашему компилятору). В программе будут созданы три круга с различными параметрами, а затем они появятся на экране. Ниже приведен листинг программы CIRCLES:

```
// circles.cpp
// круги в качестве объектов
#include "msoftcon.h"         // для функций консольной графики
//////////////////////////////////////////////////////////
class circle                  //графический объект "круг"
{
  protected:
    int xCo.yCo:              // координаты центра
    int radius:
    color fillcolor:          // цвет
    fstyle fillstyle:         // стиль заполнения
  public:                     // установка атрибутов круга
    void set(int x. int y. int r. color fc. fstyle fs)
    {
```

```
      xCo = x;
      yCo = y;
      radius = r;
      fillcolor = fc;
      fillstyle = fs;
    }
    void draw()                   // рисование круга
    {
      set_color(fillcolor);       // установка цвета и
      set_fill_style(fillstyle);  // стиля заполнения
      draw_circle(xCo,yCo,radius); // рисование круга
    }
};
//////////////////////////////////////////////////////////
int main()
{
  init_graphics();              // инициализация графики
  circle c1;                    // создание кругов
  circle c2;
  circle c3;
  // установка атрибутов кругов
  c1.set(15, 7, 5, cBLUE, X_FILL);
  c2.set(41, 12, 7, cRED, O_FILL);
  c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);
  c1.draw();                    // рисование кругов
  c2.draw();
  c3.draw();
  set_cursor_pos(1,25);         // нижний левый угол
  return 0;
}
```

Результат работы программы CIRCLES такой же, как и для программы CIRCSTRC (см. рис. 5.5). Возможно, вам будет интересно сравнить две программы. В программе CIRCLES каждый из кругов представлен в виде объекта, а не совокупностью структуры и независимой от нее функции circ_draw(), как это было в программе CIRCSTRC. Обратите внимание, что в программе CIRCLES все, что имеет отношение к кругам, то есть соответствующие данные и функции, объединены в единое целое в определении класса.

Кроме функции draw(), класс circle содержит функцию set(), имеющую пять аргументов и задающую параметры круга. Как мы увидим позже, вместо функции set() лучше использовать конструктор.

Класс как тип данных

Здесь мы рассмотрим пример, демонстрирующий применение объектов C++ в качестве переменных типа, определенного пользователем. Объекты будут представлять расстояния, выраженные в английской системе мер, описанной в главе 4. Ниже приведен листинг программы ENGLOBJ:

```
// englobj.cpp
// длины в английской системе в качестве объектов
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////////
class Distance                    // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:
    void setdist( int ft, float in ) // установка значений полей
      { feet = ft; inches = in; }
    void getdist()                // ввод полей с клавиатуры
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()               // вывод полей на экран
      { cout << feet << "\'-" << inches << '\"'; }
};
////////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist2;          // две длины
  dist1.setdist(11,6.25);         // установка значений для d1
  dist2.getdist();                // ввод значений для dist2
  // вывод длин на экран
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << endl;
  return 0;
}
```

В этой программе класс Distance содержит два поля: feet и inches. Он схож со структурой Distance, рассмотренной в главе 4, однако класс Distance имеет три метода: setdist(), предназначенный для задания значений полей объекта через передаваемые ему аргументы, getdist(), получающий эти же значения с клавиатуры, и showdist(), отображающий на экране расстояние в футах и дюймах.

Таким образом, значения полей объекта класса Distance могут быть заданы двумя способами. В функции main() мы определили две переменные типа Distance: dist1 и dist2. Значения полей для первой из них задаются с помощью функции setdist(), вызванной с аргументами 11 и 6.25, а значения полей переменной dist2 вводятся пользователем. Результат работы программы выглядит следующим образом:

```
Введите число футов: 10
Введите число дюймов: 4.75

dist1 = 11'-6.25" - задано аргументами программы
dist1 = 10'-4.75" - введено пользователем
```

Конструкторы

Пример ENGLOBJ демонстрирует два способа использования методов класса для инициализации полей объекта класса. Как правило, удобнее инициализировать поля объекта автоматически в момент его создания, а не явно вызывать в программе соответствующий метод. Такой способ инициализации реализуется

с помощью особого метода класса, называемого *конструктором*. Конструктор — это метод класса, выполняющийся автоматически в момент создания объекта.

Пример со счетчиком

В качестве примера мы создадим класс, объекты которого могут быть полезны практически для любой программы. Счетчик — это средство, предназначенное для хранения количественной меры какой-либо изменяющейся величины. Счетчик может хранить число обращений к файлу, число раз, которое пользователь нажал клавишу Enter, или количество клиентов банка. Как правило, при наступлении соответствующего события счетчик увеличивается на единицу (инкрементируется). Обращение к счетчику происходит, как правило, для того, чтобы узнать текущее значение той величины, для измерения которой он предназначен.

Допустим, что счетчик, который мы сейчас создадим, будет важной частью нашей программы, и многие из ее функций будут использовать значение этого счетчика. В процедурных языках, таких, как С, счетчик, скорее всего, был бы представлен в виде глобальной переменной. Но, как мы уже говорили в главе 1, использование глобальных переменных усложняет разработку программы и небезопасно с точки зрения несанкционированного доступа со стороны функций. Наш следующий пример, COUNTER, использует такой счетчик, значение которого может быть изменено только с помощью его собственных методов.

```
// counter.cpp
// счетчик в качестве объекта
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class Counter
{
  private:
    unsigned int count;          // значение счетчика
  public:
    Counter() : count(0)         // конструктор
      { /* пустое тело */ }
    void inc_count()             // инкрементирование счетчика
      { count++; }
    int get_count()              // получение значения счетчика
      { return count; }
};
//////////////////////////////////////////////////////////
int main()
{
  Counter c1, c2;                // определение с инициализацией
  cout <<"\nc1=" << c1.get_count(); // вывод
  cout <<"\nc2=" << c2.get_count();
  c1.inc_count();                // инкрементирование c1
  c2.inc_count();                // инкрементирование c2
  c2.inc_count();                // инкрементирование c2
  cout <<"\nc1=" << c1.get_count(); // вывод
  cout <<"\nc2=" << c2.get_count();
  cout << endl;
  return 0;
}
```

Класс Counter имеет единственное поле count типа unsigned int, поскольку значение счетчика не может быть отрицательным, и три метода: конструктор Counter(), который мы рассмотрим чуть позже, inc_count(), инкрементирующий поле count, и get_count(), возвращающий текущее значение счетчика.

Автоматическая инициализация

Когда создается объект типа Counter, нам хотелось бы, чтобы его поле count было инициализировано нулевым значением, поскольку большинство счетчиков начинают отсчет именно с нуля. Мы могли бы провести инициализацию с помощью вызова функции set_count() с аргументом, равным нулю, или создать специальный метод zero_count(), обнуляющий значение функции. Недостаток такого подхода заключается в том, что эти функции необходимо вызывать явно каждый раз при создании объекта типа Counter:

```
Counter c1;                    // при определении объекта
c1.zero_count();               // это необходимое действие
```

Подобные действия легко могут привести к неправильной работе всей программы, поскольку программисту для этого достаточно забыть проинициализировать хотя бы одну переменную после ее создания. Если в программе создается множество таких переменных, гораздо проще и надежнее было бы инициализировать их автоматически при создании. В нашем примере конструктор Counter() выполняет эти действия. Конструктор вызывается автоматически при создании каждого из объектов. Таким образом, в функции main() оператор

```
Counter c1. c2;
```

создает два объекта типа Counter. При создании каждого из них вызывается конструктор Counter(), присваивающий полю counter нулевое значение. Таким образом, кроме создания переменных, данный оператор еще присваивает их полям нулевое значение.

Имя конструктора

У конструкторов есть несколько особенностей, отличающих их от других методов класса. Во-первых, имя конструктора в точности совпадает с именем класса (в нашем примере таким именем является Counter). Таким образом, компилятор отличает конструкторы от других методов класса. Во-вторых, у конструкторов не существует возвращаемого значения. Это объясняется тем, что конструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой конструктор мог бы возвратить значение. Следовательно, указание возвращаемого значения для конструктора не имеет смысла. Отсутствие типа возвращаемого значения у конструкторов является вторым признаком, по которому компилятор может отличить их от других методов класса.

Список инициализации

Одной из наиболее часто возлагаемых на конструктор задач является инициализация полей объекта класса. Для каждого объекта класса Counter конструктор выполняет инициализацию поля count нулем. Вы, вероятно, ожидали, что это

действие будет произведено в теле конструктора приблизительно следующим образом:

```
count()
  { count = 0; }
```

Такая форма записи не рекомендуется, несмотря на то, что она не содержит ошибок. Инициализация в нашем примере происходит следующим образом:

```
count(): count(0)
{  }
```

Инициализация расположена между прототипом метода и телом функции и предварена двоеточием. Инициализирующее значение помещено в скобках после имени поля.

Если необходимо инициализировать сразу несколько полей класса, то значения разделяются запятыми, и в результате образуется *список инициализации*:

```
someClass() : m1(7), m2(33), m2(4)
{  }
```

Причины, по которым инициализация не проводится в теле конструктора, достаточно сложны. Инициализация полей с помощью списка инициализации происходит до начала исполнения тела конструктора, что в некоторых ситуациях бывает важно. Так, например, список инициализации — это единственный способ задать начальные значения констант и ссылок. В теле конструктора, как правило, производятся более сложные действия, чем обычная инициализация.

Результаты работы программы со счетчиком

В функции main() рассматриваемой нами программы создаются два объекта класса Counter с именами c1 и c2. Затем на экран выводятся значения полей каждого из объектов, которые, согласно нашей задумке, должны быть инициализированы нулевыми значениями. Далее значение счетчика c1 инкрементируется один раз, а значение счетчика c2 — два раза, и программа вновь заставляет объекты вывести значения своих полей на экран (что является в данном случае вполне корректным). Результат работы программы выглядит следующим образом:

```
c1=0
c2=0
c1=1
c2=2
```

Для того чтобы убедиться в том, что конструктор функционирует именно так, как мы описали выше, заставим его печатать сообщение во время выполнения:

```
Counter() : count(0)
  { cout << "Конструктор\n"; }
```

Теперь результат работы программы будет выглядеть следующим образом:

```
Конструктор
Конструктор
c1=0
```

```
c2=0
c1=1
c2=2
```

Как можно видеть, конструктор исполняется дважды: первый раз — для переменной c1, второй раз — для переменной c2, во время выполнения оператора

```
Counter c1, c2;
```

в функции main().

Конструкторы и собственные типы данных

Разработчики компиляторов для языков C, VB или C++ должны позаботиться о том, чтобы для любой переменной стандартного типа, которую программист определяет в своей программе, вызывался необходимый конструктор. Например, если в программе встречается определение переменной типа int, где-то должен существовать конструктор, который выделит для этой переменной четыре байта памяти. Таким образом, научившись создавать свои собственные конструкторы, мы можем выполнять задачи, с которыми сталкиваются разработчики компиляторов. Мы сделали еще один шаг на пути к созданию собственных типов данных, в чем мы скоро убедимся.

Графический пример

Давайте модифицируем пример CIRCLES так, чтобы вместо функции set() использовался конструктор. Чтобы инициализировать пять полей класса, конструктору необходимо указать пять имен и пять соответствующих им значений в списке инициализации. Приведем листинг программы CIRCTOR:

```
// circtor.cpp
// графические объекты "круг" и конструкторы
#include "msoftcon.h "          // для функций консольной графики
///////////////////////////////////////////////////////////
class circle                    // графический объект "круг"
{
  protected:
    int xCo,yCo;                // координаты центра
    int radius;
    color fillcolor;            // цвет
    fstyle fillstyle;           // стиль заполнения
  public:
    // конструктор
    circle( int x, int y, int r, color fc, fstyle fs ):
    xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
    { }
    void draw()                 // рисование круга
    {
      set_color(fillcolor);     // установка цвета и
      set_fill_style(fillstyle); // стиля заполнения
      draw_circle(xCo,yCo,radius);// вывод круга на экран
    }
};
///////////////////////////////////////////////////////////
```

```
int main()
{
  init_graphics();              // инициализация графики
  // создание кругов
  circle c1(15, 7, 5, cBLUE, X_FILL);
  circle c2(41, 12, 7, cRED, O_FILL);
  circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);
  c1.draw();                    // рисование кругов
  c2.draw();
  c3.draw();
  set_cursor_pos(1,25);        // левый нижний угол
  return 0;
}
```

Данная программа схожа с программой CIRCLES, за исключением того, что функция set() замещена конструктором. Обратите внимание, как это изменение упростило функцию main(). Вместо двух операторов, используемых для создания объекта и для его инициализации, используется один оператор, совмещающий эти два действия.

Деструкторы

Как мы уже видели, особый метод класса — конструктор — вызывается при создании объекта. Как вы, возможно, догадались, существует другая функция, автоматически вызываемая при уничтожении объекта и называемая *деструктором*. Деструктор имеет имя, совпадающее с именем конструктора (а следовательно, и класса) и предваряющееся символом ~:

```
class Foo
  {
    private:
      int data;
    public:
      Foo() : data(0)         //конструктор (имя такое же, как у класса)
        { }
     ~Foo()                   //деструктор  (то же имя, но с символом ~)
        { }
};
```

Подобно конструкторам, деструкторы не возвращают значения и не имеют аргументов, поскольку невозможно уничтожение объектов несколькими способами. Наиболее распространенное применение деструкторов — освобождение памяти, выделенной конструктором при создании объекта. Более подробно мы рассмотрим эти действия в главе 10 «Указатели», а пока использование деструкторов не будет иметь для нас большого смысла.

Объекты в качестве аргументов функций

В следующей нашей программе мы внесем улучшения в пример ENGLOBJ, а также продемонстрируем несколько новых аспектов создания классов: перегрузку конструкторов, определение методов класса вне класса и, возможно, самое важное —

использование объектов в качестве аргументов функций. Рассмотрим программу ENGLCON:

```
// englcon.cpp
// constructors.adds objects using member function
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance                 // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                      // конструктор без аргументов
    Distance() : feet(0),inches(0.0)
      { }
    // конструктор с двумя аргументами
    Distance(int ft,float in) : feet(ft),inches(in)
      { }
    void getdist()             // ввод длины пользователем
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()            // вывод длины на экран
      { cout << feet << "\'-" << inches << '\"'; }
    void add_dist( Distance, Distance ); // прототип
};
//--------------------------------------------------------
// сложение длин d2 и d3
void Distance::add_dist(Distance d2, Distance d3)
{
  inches = d2.inches + d3.inches;      // сложение дюймов
  feet = 0;                            // с возможным заемом
  if(inches >= 12.0)                   // если число дюймов больше 12.0,
  {                                    // то уменьшаем число дюймов
    inches -= 12.0;                    // на 12.0 и увеличиваем
    feet++;                            // число футов на 1
  }
  feet +=d2.feet + d3.feet;           // сложение футов
}
///////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;        // две длины
  Distance dist2(11, 6.25);     // определение и инициализация
  dist1.getdist();              // ввод dist1
  dist3.add_dist(dist1,dist2);  // dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Основной блок этой программы начинается с присвоения начальных значений полям объекта dist2 класса Distance, после чего производится его сложение

с экземпляром dist1, инициализируемым пользователем. Затем на экран выводятся все три экземпляра класса Distance:

```
Введите число футов: 17
Введите число дюймов: 5.75
dist1 = 17'-5.75"
dist2 = 11'-6.25"
dist3 = 29'-0"
```

Рассмотрим теперь, как выполняются новые действия, включенные в программу.

Перегруженные конструкторы

Было бы удобно производить инициализацию переменных класса Distance в момент их создания, то есть использовать объявления типа

```
Distance width(5, 6.25);
```

где определяется объект width, сразу же инициализируемый значениями 5 и 6.25 для футов и дюймов соответственно.

Чтобы сделать это, вызовем конструктор следующим образом:

```
Distance(int ft, float in) : feet(ft), inches(in)
{ }
```

Мы инициализируем поля feet и inches теми значениями, которые передаются конструктору в качестве аргументов.

Тем не менее, мы хотим сохранить возможность определять переменные класса Distance без инициализации, подобно тому, как мы делали в программе ENGLOBJ.

```
Distance dist1, dist2;
```

В программе ENGLOBJ не было конструктора, но определения работали без ошибок. Почему же они работали без конструктора? Это объясняется тем, что компилятор автоматически встраивает в программу конструктор без параметров, который и создает переменные класса, несмотря на то, что явного определения конструктора мы не делали. Такой конструктор без параметров называется *конструктором по умолчанию*. Если бы конструктор по умолчанию не создавался автоматически, то мы не смогли бы определять переменные классов, в которых отсутствует конструктор.

Зачастую нам хотелось бы, чтобы начальные значения полям объекта присваивались также и в конструкторе без параметров. Если возложить эту функцию на конструктор по умолчанию, то мы не сможем узнать, какими значениями были инициализированы поля. Если же для нас важно, какими значениями будут инициализироваться поля объекта класса, то нам следует явно определить конструктор. В программе ENGLCON мы поступаем подобным образом:

```
Distance() : feet(0), inches(0.0)  //конструктор по умолчанию
      { }                          //тело функции пусто, никаких действий не производится
```

Члены класса инициализируются константными значениями, в данном случае целым значением 0 для переменной feet и вещественным значением 0.0 для переменной inches. Значит, мы можем использовать объекты, инициализиру-

емые с помощью конструктора без параметров, будучи уверенными в том, что поля объекта имеют нулевые, а не другие, случайные, значения.

Теперь у нас имеется два явно определенных конструктора с одним и тем же именем Distance(), и поэтому говорят, что конструктор является *перегруженным*. Какой из этих двух конструкторов исполняется во время создания нового объекта, зависит от того, сколько аргументов используется при вызове:

```
Distance length;              //вызывает первый конструктор
Distance width(11, 6.0);      //вызывает второй конструктор
```

Определение методов класса вне класса

До сих пор мы всегда определяли методы класса внутри самого класса. На самом деле это не является обязательным. В примере ENGLCON метод add_dist() определен вне класса Distance(). Внутри определения класса содержится лишь прототип функции add_dist():

```
void add_dist(Distance, Distance);
```

Такая форма означает, что функция является методом класса, однако ее определение следует искать не внутри определения класса, а где-то в другом месте листинга.

В примере ENGLCON функция add_dist() определена позже, чем класс Distance(). Ее код взят из программы ENGLSTRC главы 4:

```
void Distance::add_dist(Distance d2, Distance d3)
{
  inches = d2.inches + d3.inches;  // сложение дюймов
  feet = 0;                        // с возможным заемом
  if(inches >= 12.0)               // если число дюймов больше 12.0,
  {                                // то уменьшаем число дюймов
    inches -= 12.0;                // на 12.0 и увеличиваем
    feet++;                        // число футов на 1
  }
  feet += d2.feet + d3.feet;       // сложение футов
}
```

Заголовок функции содержит не встречавшиеся нам ранее синтаксические элементы. Перед именем функции add_dist() стоит имя класса Distance и новый символ ::. Этот символ является знаком *операции глобального разрешения*. Такая форма записи устанавливает взаимосвязь функции и класса, к которой относится эта функция. В данном случае запись Distance::add_list() означает, что функция add_dist() является методом класса Distance. Вышесказанное проиллюстрировано на рис. 6.5.

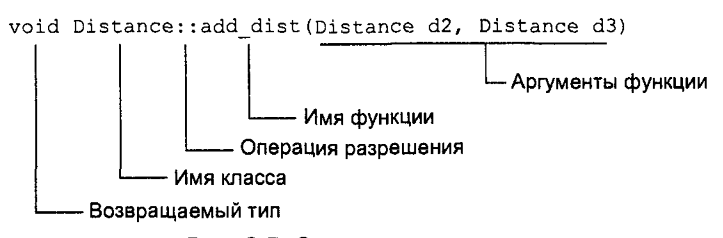


Рис. 6.5. Операция разрешения

Объекты в качестве аргументов

Рассмотрим, каким образом исполняется программа ENGLCON. Объекты dist1 и dist3 создаются при помощи конструктора по умолчанию (конструктора без аргументов), а объект dist2 — при помощи конструктора, принимающего два аргумента, значения которых инициализируют поля объекта dist2. Значения для инициализации объекта dist1 вводятся пользователем при помощи метода getdist().

Теперь мы хотим сложить значения dist1 и dist2 и присвоить результат объекту dist3. Это делается с помощью вызова функции

```
dist3.add_dist(dist1, dist2);
```

Величины, которые мы хотим сложить, передаются в качестве аргументов методу add_dist(). Синтаксис передачи объектов в функцию такой же, как и для переменных стандартных типов: на месте аргумента указывается имя объекта. Поскольку add_dist() является методом класса Distance, он имеет доступ к любым полям любого объекта класса Distance, используя операцию точки (.), например dist1.inches и dist2.feet.

Если внимательно изучить функцию add_dist(), то мы увидим несколько важных деталей, касающихся методов класса. Методу класса всегда предоставлен доступ к полям объекта, для которого он вызван: объект связан с методом операцией точки (.). Однако на самом деле методу класса доступны и другие объекты. Если задаться вопросом, к каким объектам имеет доступ метод add_dist(), то, взглянув на вызов

```
dist3.add_dist(dist1, dist2);
```

можно заметить, что, кроме объекта dist3, из которого был вызван метод add_dist(), он имеет доступ также и к объектам dist1 и dist2, поскольку они выступают в качестве его аргументов. Можно рассматривать объект dist3 как псевдоаргумент функции add_dist(): формально он не является аргументом, но функция имеет доступ к его полям. Смысл приведенного вызова можно сформулировать так: «выполнить метод add_dist() объекта dist3». Когда внутри функции происходит обращение к полям inches и feet, это означает, что на самом деле обращение происходит к полям dist3.inches и dist3.feet.

Обратите внимание на то, что функция не возвращает значения. Типом возвращаемого значения для функции add_dist() является void. Результат автоматически присваивается объекту dist3. На рис. 6.6 приведена иллюстрация сложения значений dist1 и dist2 с сохранением результата в переменной dist3.

Подводя итог вышесказанному, мы можем утверждать, что каждый вызов метода класса обязательно связан с конкретным объектом этого класса (исключением является вызов статической функции, как мы увидим позже). Метод может прямо обращаться по имени (feet и inches в данном примере) к любым, открытым и закрытым, членам этого объекта. Кроме того, метод имеет непрямой (через операцию точки, например dist1.inches и dist2.feet) доступ к членам других объектов своего класса; последние выступают в качестве аргументов метода.

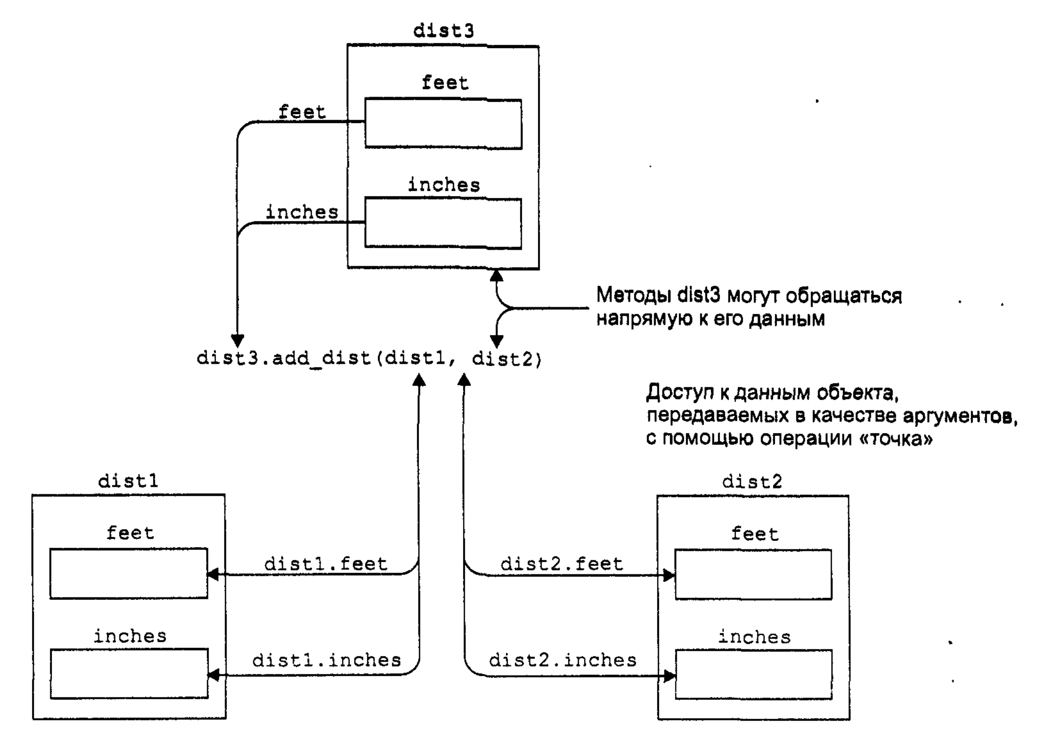


Рис. 6.6. Результат

Конструктор копирования по умолчанию

Мы рассмотрели два способа инициализации объектов. Конструктор без аргументов может инициализировать поля объекта константными значениями, а конструктор, имеющий хотя бы один аргумент, может инициализировать поля значениями, переданными ему в качестве аргументов. Теперь мы рассмотрим третий способ инициализации объекта, использующий значения полей уже существующего объекта. К вашему возможному удивлению, для этого не нужно самим создавать специальный конструктор, поскольку такой конструктор предоставляется компилятором для каждого создаваемого класса и называется *копирующим конструктором по умолчанию*. Копирующий конструктор имеет единственный аргумент, являющийся объектом того же класса, что и конструктор. Программа ECOPYCON демонстрирует использование копирующего конструктора по умолчанию:

```
// ecopycon.cpp
// инициализация объектов с помощью копирующего конструктора
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance                // длина в английской системе
```

```
{
  private:
    int feet;
    float inches;
  public:
    // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }
    // конструктора с одним аргументом нет!
    // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()            // ввод длины пользователем
    {
      cout << "\nВведите число футов "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()           // вывод длины
      { cout << feet << "\'-" << inches << '\"'; }
};
//////////////////////////////////////////////////////////
int main()
{
  Distance dist1(11, 6.25);  // конструктор с двумя аргументами
  Distance dist2(dist1);     // два конструктора с одним аргументом
  Distance dist3 = dist1;
  // вывод всех длин
  cout <<"\ndist1 = "; dist1.showdist();
  cout <<"\ndist2 = "; dist2.showdist();
  cout <<"\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Мы инициализировали объект dist1 значением 11'-6.25" при помощи конструктора с двумя аргументами. Затем мы определяем еще два объекта класса Distance с именами dist2 и dist3, оба из которых инициализируются значением объекта dist1. Возможно, вам покажется, что в данном случае должен был вызваться конструктор с одним аргументом, но поскольку аргументом являлся объект того же класса, что и инициализируемые объекты, были предприняты иные действия. В обоих случаях был вызван копирующий конструктор по умолчанию. Объект dist2 инициализирован при помощи оператора

```
Distance dist2(dist1);
```

Действие копирующего конструктора по умолчанию сводится к копированию значений полей объекта dist1 в соответствующие поля объекта dist2. Как это ни удивительно, но идентичные действия для пары объектов dist1 и dist3 выполняются при помощи оператора

```
Distance dist3 = dist1;
```

Можно подумать, что данный оператор выполняет операцию присваивания, но это не так. Здесь, как и в предыдущем случае, вызывается конструктор копирования по умолчанию. Оба оператора выполняют одинаковые действия и рав-

ноправны в использовании. Результат работы программы выглядит следующим образом:

```
dist1 = 11'-6.25"
dist2 = 11'-6.25"
dist3 = 11'-6.25"
```

Видим, что объекты dist2 и dist3 действительно имеют те же значения, что и объект dist1. В главе 11 «Виртуальные функции» мы расскажем, каким образом можно создать свой собственный копирующий конструктор с помощью перегрузки копирующего конструктора по умолчанию.

Объекты, возвращаемые функцией

В примере ENGLCON мы видели, что объекты можно передавать в функцию в качестве аргументов. Теперь мы рассмотрим, каким образом функция может возвратить объект в вызывающую программу. Мы модифицируем пример ENGLCON и в результате получим программу ENGLRET:

```
// englret.cpp
// возвращение функцией значения типа Distance
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance                 // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                      // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }                      // конструктор с двумя аргументами
    Distance(int ft,float in) : feet(ft),inches(in)
      { }
    void getdist()             // ввод длины
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()            // вывод длины
      { cout << feet << "\'-" << inches << '\"'; }
    Distance add_dist(Distance);    // сложение
};
//---------------------------------------------------------
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(Distance d2)
{
  Distance temp;                   // временная переменная
  temp.inches = inches + d2.inches; // сложение дюймов
  if(temp.inches >= 12.0)          // если сумма больше 12.0,
  {                                // то уменьшаем ее на
    temp.inches -= 12.0;           // 12.0 и увеличиваем
```

```
    temp.feet = 1;                  // число футов на 1
  }
  temp.feet += feet + d2.feet;      // сложение футов
  return temp;
}
///////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;            // две длины
  Distance dist2(11, 6.25);         // определение и инициализация dist2
  dist1.getdist();                  // ввод dist1 пользователем
  dist3 = dist1.add_dist(dist2);    // dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Программа ENGLRET очень похожа на программу ENGLCON, однако различия между ними указывают на важные аспекты работы функций с объектами.

Аргументы и объекты

В примере ENGLCON два объекта были переданы в качестве аргументов в функцию add_dist(), а результат был сохранен в объекте dist3, методом которого и являлась вызванная функция add_dist(). В программе ENGLRET в качестве аргумента в функцию add_dist() передается лишь один аргумент: объект dist2. dist2 складывается с объектом dist1, к которому относится вызываемый метод add_dist(), результат возвращается в функцию main()и присваивается объекту dist3:

```
dist3 = dist1.add_dist(dist2);
```

Действия этого оператора аналогичны действиям соответствующего оператора программы ENGLCON, однако форма записи первого более естественна, поскольку использует операцию присваивания самым обычным образом. В главе 8 «Перегрузка операций» мы увидим, как с помощью арифметической операции + можно добиться еще более простой формы записи этих же действий:

```
dist3 = dist1 + dist2;
```

Функция add_dist() из примера ENGLRET выглядит так:

```
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(Distance d2)
{
  Distance temp;                    // временная переменная
  temp.inches = inches + d2.inches; // сложение дюймов
  if(temp.inches >= 12.0)           // если сумма больше 12.0,
  {                                 // то уменьшаем ее на
    temp.inches -= 12.0;            // 12.0 и увеличиваем
    temp.feet = 1;                  // число футов на 1
  }
```

```
  temp.feet += feet + d2.feet;       // сложение футов
  return temp;
}
```

Сравнив эту функцию с одноименной функцией программы ENGLCON, вы можете заметить несколько тонкостей, отличающих одну функцию от другой. В функции из программы ENGLRET создается временный объект класса Distance. Этот объект хранит значение вычисленной суммы до тех пор, пока она не будет возвращена вызывающей программе. Сумма вычисляется путем сложения двух объектов класса Distance. Первый из объектов — dist1, по отношению к которому функция add_dist() является методом; второй объект — dist2, переданный в функцию в качестве аргумента. Обращение к его полям из функции выглядит как d2.feet и d2.inches. Результат сложения хранится в объекте temp и обращение к его полям выглядит как temp.feet и temp.inches. Значение объекта temp возвращается в вызывающую программу с помощью оператора

```
return temp;
```

Вызывающая программа main() присваивает значение, возвращенное функцией, объекту dist3. Обратите внимание на то, что значение переменной dist1 не изменяется, а всего лишь используется функцией add_dist(). На рис. 6.7 показано, каким образом значение объекта возвращается в вызывающую программу.

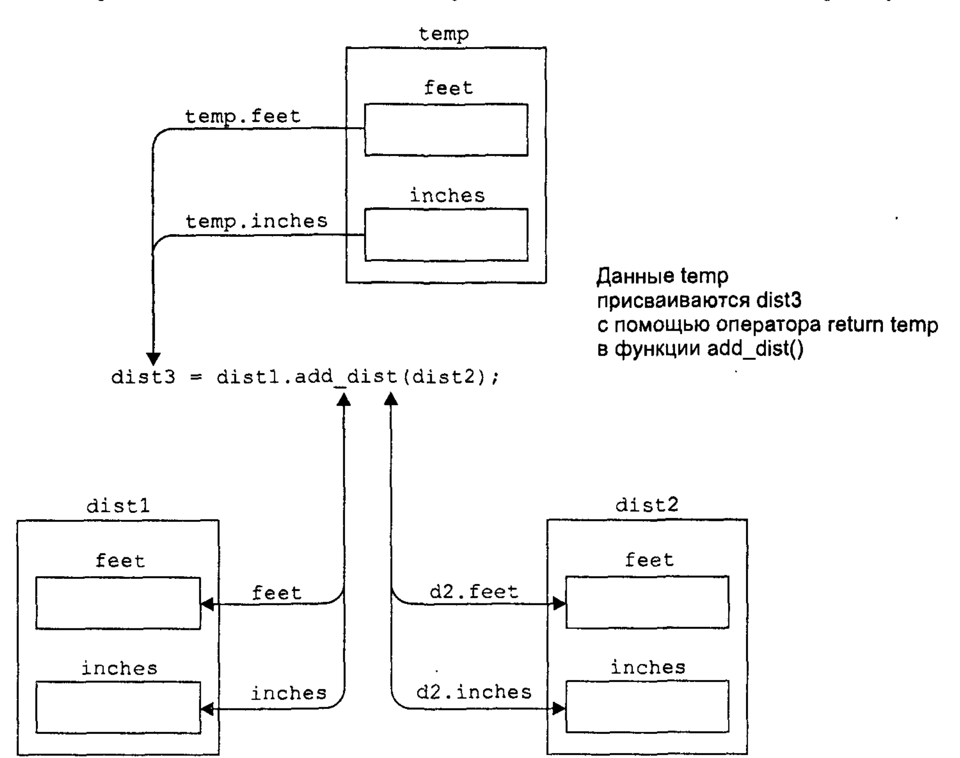


Рис. 6.7. Результат, возвращаемый из временного объекта

Пример карточной игры

В качестве примера моделирования объектами реально существующего мира рассмотрим модификацию программы CARDS, созданную в главе 4. В новой ее версии мы будем использовать объекты. Никаких дополнительных действий в программу мы вводить не будем, но сохраним практически все основные возможности старой версии.

Как и программа CARDS, ее версия CARDOBJ создает три карты фиксированной масти и достоинства, «перемешивает» их, раскладывает и предлагает вам угадать их очередность. Разница между программами состоит лишь в том, что каждая карта представлена объектом класса card.

```
// cardobj.cpp
// игральные карты в качестве объектов
#include <iostream>
using namespace std:
enum Suit { clubs. diamonds. hearts.spades }:
const int jack = 11:           // именованные достоинства карт
const int queen = 12:
const int king = 13:
const int ace = 14:
//////////////////////////////////////////////////////////
class card
{
  private:
    int number:                // достоинство карты
    Suit suit:                 // масть
  public:
    card ()                    // конструктор без аргументов
      { }
                    // конструктор с двумя аргументами
    card (int n. Suit s) : number(n). suit(s)
      { }
    void display():            // вывод карты на экран
    bool isEqual(card):        // результат сравнения карт
}:
//--------------------------------------------------------
void card::display()           // вывод карты на экран
{
  if( number>=2 && number<=10 )
    cout << number :
  else
    switch(number)
    {
      case jack:   cout << "Валет ":  break:
      case queen:  cout << "Дама ":   break:
      case king:   cout << "Король ": break:
      case ace:    cout << "Туз ":    break:
    }
  switch(suit)
  {
    case clubs:     cout << "треф"   :break:
    case diamonds:  cout << "бубен"  :break:
```

```
    case hearts:    cout << "червей" ;break;
    case spades:    cout << "пик"    ;break;
  }
}
//--------------------------------------------------------
bool card::isEqual(card c2)  // сравнение двух карт
{
  return (number==c2.number && suit==c2.suit)? true : false;
}
//////////////////////////////////////////////////////////
int main()
{
  card temp, chosen, prize;  // 3 карты
  int position;
  card card1(7, clubs);      // определение и инициализация card1
  cout << "\nКарта 1: ";
  card1.display();           // вывод card1
  card card2(jack, hearts);  // определение и инициализация card2
  cout <<  "\nКарта 2: ";
  card2.display();           // вывод card2
  card card3(ace, spades);   // определение и инициализация card3
  cout <<  "\nКарта 3: ";
  card3.display();           // вывод card3
  prize = card3;             // карту prize будет необходимо угадать
  cout << "\nМеняем местами карты 1 и 3...";
  temp = card3; card3 = card1; card1 = temp;
  cout << "\nМеняем местами карты 2 и 3...";
  temp = card3; card3 = card2; card2 = temp;
  cout << "\nМеняем местами карты 1 и 2...";
  temp = card2; card2 = card1; card1 = temp;
  cout << "\nНа какой позиции (1,2 или 3) теперь ";
  prize.display();           // угадываемая карта
  cout << "?";
  cin >> position;           // ввод позиции игроком
  switch (position)
  {                          // chosen - карта на позиции,
    case 1:chosen = card1; break; // выбранной игроком
    case 2:chosen = card2; break;
    case 3:chosen = card3; break;
  }
  if(chosen.isEqual(prize))      // сравнение карт
    cout << "Правильно! Вы выиграли!";
  else
    cout << "Неверно. Вы проиграли.";
   cout << " Вы выбрали карту ";
  chosen.display();              // вывод выбранной карты
  cout << endl;
  return 0;
}
```

Класс card содержит два конструктора. Первый из них не имеет аргументов и используется функцией main() для создания объектов temp, chosen и prize, которые не инициализируются при создании. Другой конструктор имеет два аргумента, и с его помощью создаются объекты card1, card2 и card3, инициализируемые определенными значениями. Кроме конструкторов, класс card содержит два метода, оба из которых определены вне класса.

Функция display() не имеет аргументов; ее задача заключается в отображении на экране параметров той карты, к которой она применяется. Так, оператор

```
chosen.display();
```

отображает карту, которую ввел пользователь.

Функция isEqual() проверяет, совпадает ли карта, введенная пользователем, с той, которая передается функции в качестве аргумента. Для сравнения используется условная операция. Функция могла бы быть реализована с помощью ветвления if...else:

```
if(number==c2.number && suit==c2.suit)
  return true;
else
  return false;
```

однако условная операция обеспечивает более краткую форму записи.

Название c2 аргумента функции isEqual() указывает на то, что сравниваются две карты: первая из карт представлена объектом, методом которой является функция isEqual(). Выражение

```
if ( chosen.isEqual(prize) )
```

в функции main() сравнивает карту chosen с картой prize.

Если игрок не угадал карту, то результат работы программы будет приблизительно таким:

```
Карта 1: 7 треф
Карта 2: валет червей
Карта 3: туз пик
Меняем местами карты 1 и 3j
Меняем местами карты 2 и 3j
Меняем местами карты 1 и 2j
На какой позиции (1,2 или 3) теперь туз пик? 1
Неверно. Вы проиграли. Вы выбрали карту 7 треф
```

Структуры и классы

Все примеры, рассмотренные нами до сих пор, подтверждали негласный принцип: структуры предназначены для объединения данных, а классы — для объединения данных и функций. На самом деле, в большинстве ситуаций можно использовать структуры так же, как и классы. Формально разница между структурами и классами заключается лишь в том, что по умолчанию все члены класса являются скрытыми, а все члены структуры — открытыми.

Формат, который мы использовали при определении классов, выглядит примерно так:

```
class foo
{
  private:
    int data1;
```

```
  public:
    void func();
};
```

Поскольку ключевое слово private для членов классов подразумевается по умолчанию, указывать его явно не обязательно. Можно определять класс более компактным способом:

```
class foo
{
    int data1;
  public:
    void func();
};
```

В этом случае поле data1 сохранит свою закрытость. Многие программисты предпочитают второй стиль из-за его краткости. Мы же придерживаемся первого стиля, поскольку он более понятен.

Если вы хотите при помощи структуры выполнять те же действия, что и с использованием класса, вы можете отменить действие принятого по умолчанию ключевого слова public словом private и расположить открытые поля структуры до слова private, а закрытые поля — после слова private:

```
struct foo
{
  void func();
  private:
  int data1;
};
```

Тем не менее, чаще всего программисты не используют структуры таким образом, а придерживаются правила, о котором шла речь в начале: структуры предназначены для объединения данных, а классы — для объединения данных и функций.

Классы, объекты и память

Возможно, что в процессе изучения классов и объектов у вас сформировалось представление об объекте как о копии класса с точки зрения внутренней структуры. Это представление адекватно лишь в первом приближении. Оно акцентировано на том, что объекты являются самодостаточными программными единицами, созданными по образцу, представленному определением класса. Возможно, здесь уместна аналогия с машинами, сходящими с конвейера, каждая из которых построена согласно стандарту своей модели. Машины в этом смысле схожи с объектами, а модели — с классами.

На самом деле все устроено несколько сложнее. Верно, что каждый объект имеет собственные независимые поля данных. С другой стороны, все объекты одного класса используют одни и те же методы. Методы класса создаются и помещаются в память компьютера всего один раз — при создании класса.

Это вполне логически оправданно: нет никакого смысла держать в памяти копии методов для каждого объекта данного класса, поскольку у всех объектов методы одинаковы. А поскольку наборы значений полей у каждого объекта свои, поля объектов не должны быть общими. Это значит, что при создании объектов каждый из наборов данных занимает определенную совокупность свободных мест в памяти. Иллюстрация вышесказанного приведена на рис. 6.8.

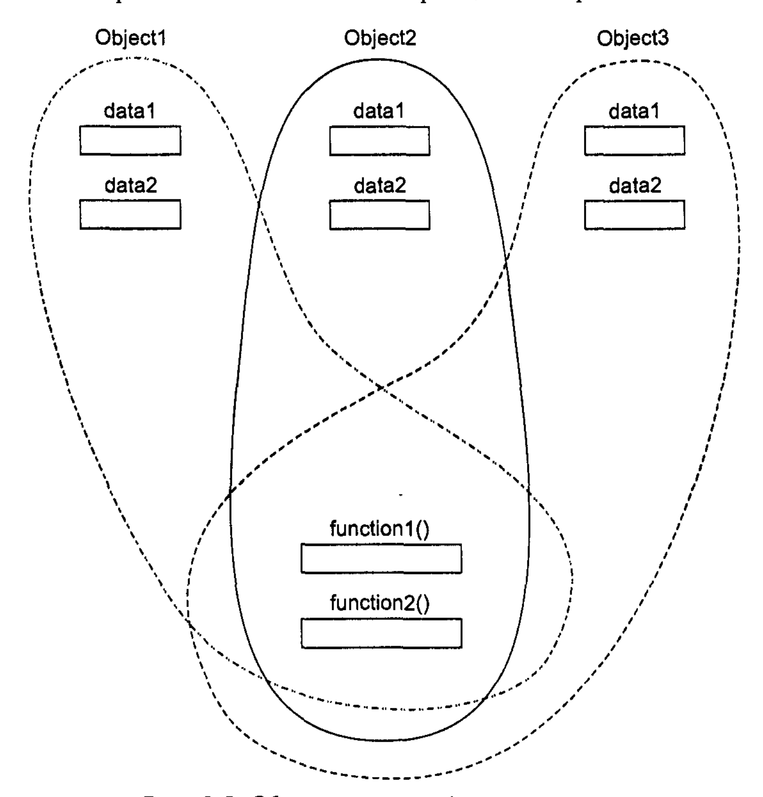


Рис. 6.8. Объекты, данные, функции и память

В примере SMALLOBJ, приведенном в начале этой главы, были созданы два объекта типа smallobj, что означало наличие двух полей с именем somedata в памяти. При этом функции setdata() и showdata() располагались в памяти в единственном экземпляре и совместно использовались обоими объектами класса. Между объектами не возникает конкуренции за общий ресурс (по крайней мере, в системах с последовательной обработкой), поскольку в любой момент времени выполняется не более одной функции.

В большинстве ситуаций вам не нужно использовать тот факт, что для всех объектов класса методы существуют в единственном экземпляре. Разумеется, гораздо проще представить себе объекты, которые включают в себя как данные, так и функции. Однако в некоторых ситуациях, например при оценке размера программы, необходимо знать, какова ее внутренняя организация.

Статические данные класса

Познакомившись с тем, что каждый объект класса содержит свои собственные данные, теперь мы должны углубить свое понимание данной концепции. Если поле данных класса описано с ключевым словом static, то значение этого поля будет одинаковым для всех объектов данного класса. Статические данные класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение. Статическое поле по своим характеристикам схоже со статической переменной: оно видимо только внутри класса, но время его жизни совпадает со временем жизни программы. Таким образом, статическое поле существует даже в том случае, когда не существует ни одного объекта класса (обсуждение статических переменных функции можно найти в главе 5). Тем не менее, в отличие от статической переменной функции, предназначенной для сохранения значения между вызовами, статическая переменная класса используется для хранения данных, совместно используемых объектами класса.

Применение статических полей класса

Для того чтобы понять, зачем могут использоваться статические переменные класса, представьте себе следующую ситуацию. Допустим, вам необходимо, чтобы объект вашего класса располагал информацией, сколько еще объектов этого же класса существует на данный момент времени в памяти. Другими словами, если вы моделируете автомобильную гонку, и объекты — это гоночные машины, то вам нужно, чтобы каждый гонщик знал о том, сколько всего автомобилей участвует в гонке. В этом случае можно включить в класс статическую переменную count. Эта переменная будет доступна всем объектам класса, и все они будут видеть одно и то же ее значение.

Пример использования статических полей класса

Следующий пример, STATDAT, иллюстрирует простое применение статического поля класса:

```
// statdata.cpp
// статические данные класса
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class foo
{
  private:
    static int count;       // общее поле для всех объектов
                            // (в смысле "объявления")
  public:
    foo()                   // инкрементирование при создании объекта
      { count++; }
    int getcount()          // возвращает значение count
      { return count; }
};
//---------------------------------------------------------
```

```
int foo::count = 0;          // *определение* count
//////////////////////////////////////////////////////////
int main()
{
  foo f1, f2, f3;            // создание трех обьектов
  // каждый обьект видит одно и то же значение
  cout << "Число обьектов: " << f1.getcount() << endl;
  cout << "Число обьектов: " << f2.getcount() << endl;
  cout << "Число обьектов: " << f3.getcount() << endl;
  return 0;
}
```

В этом примере класс foo содержит единственное поле count, имеющее тип static int. Конструктор класса инкрементирует значение поля count. В функции main() мы определяем три объекта класса foo. Поскольку конструктор в этом случае вызывается трижды, инкрементирование поля count также происходит трижды. Метод getcount() возвращает значение count. Мы вызываем этот метод для каждого из объектов, и во всех случаях он возвращает одну и ту же величину:

```
Число обьектов: 3
Число обьектов: 3
Число обьектов: 3
```

Если бы мы использовали не статическое, а автоматическое поле count, то конструктор увеличивал бы на единицу значение этого поля для каждого объекта, и результат работы программы выглядел бы следующим образом:

```
Число обьектов: 1
Число обьектов: 1
Число обьектов: 1
```

Статические поля класса применяются гораздо реже, чем автоматические, однако существует немало ситуаций, где их использование удобно. Сравнение статических и автоматических полей класса проиллюстрировано на рис. 6.9.

Раздельное объявление и определение полей класса

Определение статических полей класса происходит не так, как для обычных полей. Обычные поля объявляются (компилятору сообщается имя и тип поля) и определяются (компилятор выделяет память для хранения поля) при помощи одного оператора. Для статических полей эти два действия выполняются двумя разными операторами: объявление поля находится внутри определения класса, а определение поля, как правило, располагается вне класса и зачастую представляет собой определение глобальной переменной.

Для чего используется такая двойственная форма? Если бы определение статического поля класса находилось внутри класса (как и предполагалось в ранних версиях C++), то это нарушило бы принцип, в соответствии с которым определение класса не должно быть связано с выделением памяти. Поместив определение статического поля вне класса, мы обеспечили однократное выделение памяти под это поле до того, как программа будет запущена на выполнение и статиче-

ское поле в этом случае станет доступным всему классу. Каждый объект класса уже не будет обладать своим собственным экземпляром поля, как это должно быть с полями автоматического типа. В этом отношении можно усмотреть аналогию статического поля класса с глобальными переменными.

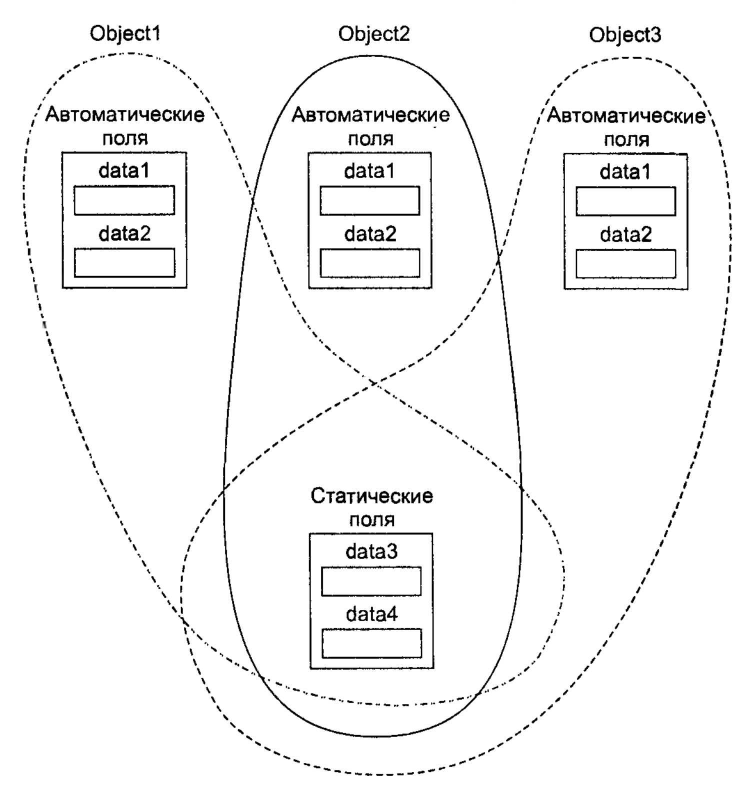


Рис. 6.9. Статические и автоматические поля класса

Работая со статическими данными класса, легко совершить ошибки, которые компилятор будет не в силах распознать. Если вы объявите статическое поле класса, но забудете его определить, компилятор не выдаст предупреждающего сообщения. Ваша программа будет считаться корректной до тех пор, пока редактор связей не обнаружит ошибку и не выдаст сообщение о том, что вы пытаетесь обратиться к необъявленной глобальной переменной. Это произойдет в том случае, если вы забудете указать имя класса при объявлении, например foo:: в программе STATDATA.

const и классы

Мы рассмотрели несколько примеров использования модификатора const для того, чтобы защитить значения переменных от изменения. В главе 5 мы также использовали ключевое слово const для того, чтобы функция не могла изменить

значение аргумента, передаваемого ей по ссылке. Теперь, когда мы уже немного знакомы с классами, рассмотрим еще один способ применения модификатора const: с методами класса, их аргументами, а также с объектами класса.

Константные методы

Константные методы отличаются тем, что не изменяют значений полей своего класса. Рассмотрим это на примере под названием CONSTFU:

```
// constfu.cpp
// применение константных методов
class aClass
{
  private:
    int alpha;
  public:
    void nonFunc()            // неконстантный метод
      { alpha = 99; }         // корректно
    void conFunc()const       // константный метод
      { alpha = 99; }         // ошибка: нельзя изменить значение поля
};
```

Обычный метод nonFunc() может изменить значение поля alpha, а константный метод conFunc() не может. Если со стороны последнего будет предпринята попытка изменить поле alpha, компилятор выдаст сообщение об ошибке.

Для того чтобы сделать функцию константной, необходимо указать ключевое слово const после прототипа функции, но до начала тела функции. Если объявление и определение функции разделены, то модификатор const необходимо указывать дважды — как при объявлении функции, так и при ее определении. Те методы, которые лишь считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения полей объектов класса.

Использование константных функций помогает компилятору обнаруживать ошибки, а также указывает читающему листинг, что функция не изменяет значений полей объектов. С помощью константных функций можно создавать и использовать константные объекты, как мы увидим позже.

Пример класса Distance

Для того чтобы не включать множество нововведений в одну программу, мы до сих пор не использовали в наших примерах константные методы. Однако существует немалое количество ситуаций, где применение константных методов было бы весьма полезным. Например, метод showdist() класса Distance, неоднократно фигурировавшего в наших примерах, следовало сделать константным, потому что он не изменяет (и не должен изменять!) полей того объекта, для которого он вызывается. Он предназначен лишь для вывода текущих значений полей на экран.

Аналогично, метод add_dist() программы ENGLRET не должен изменять данных, хранящихся в объекте, из которого он вызывается. Значение этого объекта должно складываться с тем значением, которое метод принимает в качестве аргумента, а полученный результат возвращается вызывающей программе. Мы внесли изме-

нения в программу ENGLRET, сделав два указанных метода константными. Обратите внимание на то, что модификаторы const появились как в объявлениях, так и в определениях методов.

```
// engConst.cpp
// константные методы и константные аргументы
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance                  // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                       // конструктор без аргументов
    Distance() : feet(0), inches(0.0)
      { }                       // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()              // ввод длины пользователем
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
    void showdist()const        // вывод длины
      { cout << feet << "\'-" << inches <<'\"'; }
    Distance add_dist(const Distance&) const; // сложение
};
//---------------------------------------------------------
// сложение данного объекта с d2, возврат суммы
Distance Distance::add_dist(const Distance& d2) const
{
  Distance temp;                // временная переменная
  // feet = 0;                  // Ошибка: нельзя изменить поле
  // d2.feet = 0;               // Ошибка: нельзя изменить d2
  temp.inches = inches + d2.inches; // сложение дюймов
  if( temp.inches >= 12.0 )     // если сумма превышает 12.0,
  {                             // то уменьшаем ее на 12.0
    temp.inches -= 12.0;        // и увеличиваем число футов
    temp.feet = 1;              // на 1
  }
  temp.feet += feet + d2.feet;  // сложение футов
  return temp;
}
///////////////////////////////////////////////////////////
int main()
{
  Distance dist1, dist3;        // две длины
  Distance dist2(11, 6.25);     // определение и инициализация dist2
  dist1.getdist();              // ввод dist1
  dist3 = dist1.add_dist(dist2);//dist3 = dist1 + dist2
  // вывод всех длин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

В этом примере обе функции showdist() и add_dist() являются константными. В теле функции add_dist() первый из закомментированных операторов, feet=0, демонстрирует, что компилятор выдаст ошибку при попытке изменения константной функцией полей объекта, из которого она вызывалась.

Константные аргументы методов

В главе 5 мы говорили о том, что если вы хотите передать аргумент в функцию по ссылке и в то же время защитить его от изменения функцией, необходимо сделать этот аргумент константным при объявлении и определении функции. Методы классов в этом отношении не являются исключениями. В программе ENGCONST мы передаем аргумент в функцию add_dist() по ссылке, но хотим быть уверенными в том, что функция не изменит значения этого аргумента, в качестве которого при вызове выступает переменная dist2 функции main(). Для этого параметр d2 функции add_dist() указывается с модификатором const в ее объявлении и определении. Второй из закомментированных операторов показывает, что компилятор выдаст ошибку при попытке функции add_dist() изменить значение своего аргумента.

Константные объекты

В нескольких наших примерах мы видели, что ключевое слово const можно применять для защиты от изменения значений переменных стандартных типов, таких, как, например, int. Оказывается, аналогичным способом мы можем применять модификатор const и для объектов классов. Если объект класса объявлен с модификатором const, он становится недоступным для изменения. Это означает, что для такого объекта можно вызывать только константные методы, поскольку только они гарантируют, что объект не будет изменен. В качестве примера рассмотрим программу CONSTOBJ.

```
//constObj.cpp
//constant Distance objects
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance              // длина в английской системе
{
  private:
    int feet;
    float inches;
  public:                   // конструктор с двумя аргументами
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void getdist()          // неконстантный метод
    {
      cout << "\nВведите число футов: "; cin >> feet;
      cout << "Введите число дюймов: "; cin >> inches;
    }
void showdist() const       // константный метод
  { cout << feet << "\'-" << inches << '\"'; }
};
```

```
////////////////////////////////////////////////////////////
int main()
{
  const Distance football(300,0);
  //football.getdist():      // ошибка: метод getdist() неконстантный
  cout << " Длина поля равна ";
  football.showdist():       // корректно
  cout << endl:
  return 0:
}
```

Футбольное поле в американском футболе имеет длину ровно 300 футов. Если бы нам было необходимо использовать в программе величину, выражающую длину футбольного поля, то для нас имело бы смысл сделать эту величину константной. Именно это и сделано в программе CONSTOBJ в отношении объекта football. Теперь для этого объекта можно вызывать только константные методы, например showdist(). Вызовы неконстантных методов, таких, как getdist(), который присваивает полям объекта значения, вводимые пользователем, являются некорректными. Таков механизм, обеспечивающий требуемую константность объекта football.

Когда вы создаете класс, всегда является хорошим стилем объявлять константными функции, не изменяющие полей объектов класса. Это позволяет тому, кто использует данный класс, создавать константные объекты класса. Из этих объектов могут вызываться только константные функции. Использование модификатора const упрощает компилятору задачу оказания вам своевременной помощи в процессе создания программы.

Зачем нужны классы?

Теперь, когда вы имеете представление о классах и объектах, вас, наверное, интересует ответ на вопрос, какова выгода от их использования, тем более что на наших примерах мы убеждались: для выполнения определенных действий программе не обязательно использовать классы — идентичные действия выполнялись при помощи процедурного подхода.

Первым достоинством объектно-ориентированного программирования является то, что оно обеспечивает значительное сходство между объектами реального мира, моделируемыми программой, и объектами классов C++. Мы видели, как объекты классов представляли такие реально существующие вещи, как <...>, игральные карты, геометрические фигуры и т. д. Вся информация, касающаяся <...> вообще, описана при создании класса; там же описаны и все методы, то есть действия, которые можно производить с этой информацией. Это облегчает концептуализацию решаемой задачи. Вы выделяете в задаче те элементы, которые целесообразно представлять в виде объектов, а затем помещаете все данные и функции, связанные с этим объектом, внутрь самого объекта. Если объект представляет игральную карту, то с помощью описания класса вы наделяете объект полями для хранения достоинства карты, ее масти, а также функциями,

выполняющими действия по заданию этих величин, их изменению, выводу на экран, сравнению с заданными значениями и т. д.

В процедурной программе, напротив, моделирование объектов реального мира возлагается на глобальные переменные и функции. Они не способны вместе образовать легко воспринимаемую модель.

Иногда при решении задачи бывает не столь очевидно, какие элементы реального мира следует представить в качестве объектов. Например, если вы пишете компьютерный вариант игры в шахматы, то что нужно представить в качестве объекта: шахматистов, клетки шахматной доски или игровую позицию целиком?

Небольшие программы, такие, какими являются большинство наших примеров, можно создавать методом «проб и ошибок». Можно представить задачу в виде объектов каким-либо одним способом и написать определения классов для этих объектов. Если такое представление устроит вас, то вы можете продолжить разработку созданных классов. В противном случае следует выбрать другой способ представления задачи в виде совокупности объектов и повторить операцию создания классов. Чем больше опыта вы наберете, создавая объектно-ориентированные программы, тем проще вам будет представлять новые задачи в терминах ООП.

При создании больших программ метод проб и ошибок оказывается малоприменимым. Для анализа поставленной задачи и создания классов и объектов, подходящих для ее решения, используется объектно-ориентированная разработка (OOP). Мы обсудим объектно-ориентированную разработку в главе 16, которая целиком посвящена аспектам ее применения.

Некоторые преимущества объектно-ориентированного программирования не столь очевидны. Вспомните о том, что ООП было создано, чтобы бороться со сложностью больших программ. Программы небольшого размера, подобные созданным нами, в меньшей степени нуждаются в объектно-ориентированной организации. Чем больше программа, тем ощутимей эффект от использования ООП. Но даже для программ маленького размера применение объектно-ориентированного подхода способно принести определенную выгоду: например, по сравнению с процедурными программами увеличивается эффективность компилятора при поиске и выявлении ошибок концептуального уровня.

Резюме

Класс представляет собой образ, определяющий структуру своих объектов. Объекты включают в себя как данные, так и функции, предназначенные для их обработки. И данные, и функции могут быть определены как закрытые, что означает их доступность только для членов данного класса, и как открытые, то есть доступные любой функции программы. Закрытость члена класса задается ключевым словом private, а открытость — ключевым словом public.

Методом класса называется функция, являющаяся членом этого класса. Методы класса, в отличие от других функций, имеют доступ к закрытым членам класса.

Конструктор — это метод класса, имя которого совпадает с именем класса и который выполняется каждый раз при создании нового объекта. Конструктор не имеет типа возвращаемого значения, однако может принимать аргументы. Часто конструкторы применяются для инициализации создаваемых объектов класса. Конструкторы допускают перегрузку, поэтому возможна инициализация объекта несколькими способами.

Деструктор — это метод класса, именем которого является имя класса, предваренное символом ~. Вызов деструктора производится при уничтожении объекта. Деструктор не имеет ни аргументов, ни возвращаемого значения.

В памяти компьютера каждый объект имеет свои собственные участки, хранящие значения полей этого объекта, но методы класса хранятся в памяти в единственном экземпляре. Поле класса также можно сделать единым для всех объектов данного класса, описав его при определении класса с ключевым словом static.

Одной из главных причин использования объектно-ориентированного программирования является его возможность с высокой степенью точности моделировать объекты реального мира. Иногда бывает непросто представить решаемую задачу в терминах объектов и классов. Для небольших программ можно последовательно перебрать возможные способы такого представления. Для более серьезных и больших проектов применяются другие методы.

Вопросы

Ответы на перечисленные ниже вопросы можно найти в приложении Ж.

1. Для чего необходимо определение класса?
2. ________ имеет такое же отношение к ________, как стандартный тип данных к переменной этого типа.
3. В определении класса члены класса с ключевым словом private доступны:
   а) любой функции программы;
   б) в случае, если вам известен пароль;
   в) методам этого класса;
   г) только открытым членам класса.
4. Напишите определение класса leverage, включающего одно закрытое поле типа int с именем crowbar и одним открытым методом с прототипом void pry().
5. Истинно ли следующее утверждение: поля класса должны быть закрытыми?
6. Напишите оператор, создающий объект lever1 класса leverage, описанного в вопросе 4.
7. Операция точки (операция доступа к члену класса) объединяет следующие два элемента (слева направо):

а) член класса и объект класса;

б) объект класса и класс;

в) класс и член этого класса;

г) объект класса и член этого класса.

8. Напишите оператор, который вызовет метод pry() объекта lever1 (см. вопросы 4 и 6).
9. Методы класса, определенные внутри класса, по умолчанию __________.
10. Напишите метод getcrow() для класса leverage (см. вопрос 4), который будет возвращать значение поля crowbar. Метод следует определить внутри определения класса.
11. Конструктор вызывается автоматически в момент __________ объекта.
12. Имя конструктора совпадает с именем _______ .
13. Напишите конструктор, который инициализирует нулевым значением поле crowbar класса leverage (см. вопрос 4). Конструктор следует определить внутри определения класса.
14. Верно или неверно следующее утверждение: класс может иметь более одного конструктора с одним и тем же именем?
15. Методу класса всегда доступны данные:

    а) объекта, членом которого он является;

    б) класса, членом которого он является;

    в) любого объекта класса, членом которого он является;

    г) класса, объявленного открытым.
16. Предполагая, что метод getcrow(), описанный в вопросе 10, определен вне класса, объявите этот метод внутри класса.
17. Напишите новую версию метода getcrow(), описанного в вопросе 10, которая определяется вне класса.
18. Единственным формальным различием между структурами и классами в C++ является то, что ________________.
19. Пусть определены три объекта класса. Сколько копий полей класса содержится в памяти? Сколько копий методов функций?
20. Посылка сообщения объекту эквивалентна ___________.
21. Классы полезны потому, что:

    а) не занимают памяти, если не используются;

    б) защищают свои данные от доступа со стороны других классов;

    в) собирают вместе все аспекты, касающиеся отдельной вещи;

    г) адекватно моделируют объекты реального мира.
22. Истинно ли следующее утверждение: существует простой, но очень точный метод, позволяющий представлять решаемую задачу в виде совокупности объектов классов?

23. Константный метод, вызванный для объекта класса:

а) может изменять как неконстантные, так и константные поля;

б) может изменять только неконстантные поля;

в) может изменять только константные поля;

г) не может изменять как неконстантные, так и константные поля.

24. Истинно ли следующее утверждение: объект, объявленный как константный, можно использовать только с помощью константных методов?

25. Напишите объявление (не определение) функции типа const void с именем aFunc(), которая принимает один константный аргумент jerry типа float.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Создайте класс Int, имитирующий стандартный тип int. Единственное поле этого класса должно иметь тип int. Создайте методы, которые будут устанавливать значение поля, равным нулю, инициализировать его целым значением, выводить значение поля на экран и складывать два значения типа Int.

Напишите программу, в которой будут созданы три объекта класса Int, два из которых будут инициализированы. Сложите два инициализированных объекта, присвойте результат третьему, а затем отобразите результат на экране.

*2. Представьте пункт для взимания платежей за проезд по автостраде. Каждая проезжающая машина должна заплатить за проезд 50 центов, однако часть машин платит за проезд, а часть проезжает бесплатно. В кассе ведется учет числа проехавших машин и суммарная выручка от платы за проезд.

Создайте модель такой кассы с помощью класса tollBooth. Класс должен содержать два поля. Одно из них, типа unsigned int, предназначено для учета количества проехавших автомобилей, а второе, имеющее тип double, будет содержать суммарную выручку от оплаты проезда. Конструктор должен инициализировать оба поля нулевыми значениями. Метод payingCar() инкрементирует число машин и увеличивает на 0,50 суммарную выручку. Другой метод, nopayCar(), увеличивает на единицу число автомобилей, но оставляет без изменения выручку. Метод display() выводит оба значения на экран. Там, где это возможно, сделайте методы константными.

Создайте программу, которая продемонстрирует работу класса. Программа должна предложить пользователю нажать одну клавишу для того, чтобы сымитировать заплатившего автолюбителя, и другую клавишу, чтобы сымитировать недобросовестного водителя. Нажатие клавиши Esc должно привести к выдаче текущих значений количества машин и выручки и завершению программы.

*3. Создайте класс с именем time, содержащий три поля типа int, предназначенные для хранения часов, минут и секунд. Один из конструкторов класса должен инициализировать поля нулевыми значениями, а другой конструктор — заданным набором значений. Создайте метод класса, который будет выводить значения полей на экран в формате 11:59:59, и метод, складывающий значения двух объектов типа time, передаваемых в качестве аргументов.

   В функции main() следует создать два инициализированных объекта (подумайте, должны ли они быть константными) и один неинициализированный объект. Затем сложите два инициализированных значения, а результат присвойте третьему объекту и выведите его значение на экран. Где возможно, сделайте методы константными.

4. Создайте класс employee, используя упражнение 4 главы 4. Класс должен включать поле типа int для хранения номера сотрудника и поле типа float для хранения величины его оклада. Методы класса должны позволять пользователю вводить и отображать данные класса. Напишите функцию main(), которая запросит пользователя ввести данные для трех сотрудников и выведет полученную информацию на экран.

5. Взяв в качестве основы структуру из упражнения 5 главы 4, создайте класс date. Его данные должны размещаться в трех полях типа int: month, day и year. Метод класса getdate() должен принимать значение для объекта в формате 12/31/02, а метод showdate() — выводить данные на экран.

6. Расширьте содержание класса employee из упражнения 4, включив в него класс date и перечисление etype (см. упражнение 6 главы 4). Объект класса date будет использоваться для хранения даты приема сотрудника на работу. Перечисление будет использовано для хранения статуса сотрудника: лаборант, секретарь, менеджер и т. д. Последние два поля данных должны быть закрытыми в определении класса employee, как и номер и оклад сотрудника. Вам будет необходимо разработать методы getemploy() и putemploy(), предназначенные соответственно для ввода и отображения информации о сотруднике. Возможно, при создании методов вам понадобится ветвление switch для работы с перечисляемым типом etype. Напишите функцию main(), которая попросит пользователя ввести данные о трех сотрудниках, а затем выведет эти данные на экран.

7. В морской навигации координаты точки измеряются в градусах и минутах широты и долготы. Например, координаты бухты Папити на о. Таити равны 149 градусов 34.8 минут восточной долготы и 17 градусов 31.5 минут южной широты. Это записывается как 149°34.8' W, 17°31.5' S. Один градус равен 60 минутам (устаревшая система также делила одну минуту на 60 секунд, но сейчас минуту делят на обычные десятичные доли). Долгота измеряется величиной от 0 до 180 градусов восточнее или западнее Гринвича. Широта принимает значения от 0 до 90 градусов севернее или южнее экватора.

Создайте класс angle, включающий следующие три поля: типа int для числа градусов, типа float для числа минут и типа char для указания направления (N, S, E или W). Объект этого класса может содержать значение как широты, так и долготы. Создайте метод, позволяющий ввести координату точки, направление, в котором она измеряется, и метод, выводящий на экран значение этой координаты, например 179°59.9' E. Кроме того, напишите конструктор, принимающий три аргумента. Напишите функцию main(), которая сначала создает переменную с помощью трехаргументного конструктора и выводит ее значение на экран, а затем циклически запрашивает пользователя ввести значение координаты и отображает введенное значение на экране. Для вывода символа градусов (°) можно воспользоваться символьной константой '\xF8'.

8. Создайте класс, одно из полей которого хранит «порядковый номер» объекта, то есть для первого созданного объекта значение этого поля равно 1, для второго созданного объекта значение равно 2 и т. д.

   Для того чтобы создать такое поле, вам необходимо иметь еще одно поле, в которое будет записываться количество созданных объектов класса (это означает, что последнее поле должно относиться не к отдельным объектам класса, а ко всему классу в целом. Вспомните, какое ключевое слово необходимо при описании такого поля.). Каждый раз при создании нового объекта конструктор может получить значение этого поля и в соответствии с ним назначить объекту индивидуальный порядковый номер.

   В класс следует включить метод, который будет выводить на экран порядковый номер объекта. Создайте функцию main(), в которой будут созданы три объекта, и каждый объект выведет на экран свой порядковый номер, например: **Мой порядковый номер: 2** и т. п.

9. На основе структуры fraction из упражнения 8 главы 4 создайте класс fraction. Данные класса должны быть представлены двумя полями: числителем и знаменателем. Методы класса должны получать от пользователя значения числителя и знаменателя дроби в форме 3/5 и выводить значение дроби в этом же формате. Кроме того, должен быть разработан метод, складывающий значения двух дробей. Напишите функцию main(), которая циклически запрашивает у пользователя ввод пары дробей, затем складывает их и выводит результат на экран. После каждой такой операции программа должна спрашивать пользователя, следует ли продолжать цикл.

10. Создайте класс с именем ship, который будет содержать данные об учетном номере корабля и координатах его расположения. Для задания номера корабля следует использовать механизм, аналогичный описанному в упражнении 8. Для хранения координат используйте два поля типа angle (см. упражнение 7). Разработайте метод, который будет сохранять в объекте данные о корабле, вводимые пользователем, и метод, выводящий данные о корабле на экран. Напишите функцию main(), создающую три объекта класса ship, затем запрашивающую ввод пользователем информации о каждом из кораблей и выводящую на экран всю полученную информацию.

11. Модифицируйте калькулятор, созданный в упражнении 12 главы 5 так, чтобы вместо структуры fraction использовался одноименный класс. Класс должен содержать методы для ввода и вывода данных объектов, а также для выполнения арифметических операций. Кроме того, необходимо включить в состав класса функцию, приводящую дробь к несократимому виду. Функция должна находить наибольший общий делитель числителя и знаменателя и делить числитель и знаменатель на это значение. Код функции, названной lowterms(), приведен ниже:

```
void fraction::lowterms()    // сокращение дроби
{
long tnum, tden, temp, gcd;
tnum = labs(num);            // используем неотрицательные
tden = labs(den);            // значения (нужен cmath)
if( tden == 0 )              // проверка знаменателя на 0
  { cout << "Недопустимый знаменатель!"; exit(1); }
else if( tnum == 0 )         // проверка числителя на 0
  { num=0; den = 1; return; }
// нахождение наибольшего общего делителя
while(tnum !=0)
{
  if( tnum < tden )          // если числитель больше знаменателя,
    { temp=tnum; tnum=tden; tden=temp; } //меняем их местами
  tnum = tnum - tden;        // вычитание
}
gcd = tden;                  // делим числитель и знаменатель на
num = num / gcd;             // полученный наибольший общий делитель
den = den / gcd;
}
```

Можно вызывать данную функцию в конце каждого метода, выполняющего арифметическую операцию, либо непосредственно перед выводом на экран результата. Кроме перечисленных методов, вы можете включить в класс конструктор с двумя аргументами, что также будет полезно.

12. Используйте преимущество ООП, заключающееся в том, что однажды созданный класс можно помещать в другие программы. Создайте новую программу, которая будет включать класс fraction, созданный в упражнении 11. Программа должна выводить аналог целочисленной таблицы умножения для дробей. Пользователь вводит знаменатель, а программа должна подобрать всевозможные целые значения числителя так, чтобы значения получаемых дробей находились между 0 и 1. Дроби из получившегося таким образом набора перемножаются друг с другом во всевозможных комбинациях, в результате чего получается таблица следующего вида (для знаменателя, равного 6):

```
1/61/31/22/35/6
-----------------------------------------------------
1/61/361/181/121/95/36
1/31/181/91/62/95/18
1/21/121/61/41/35/12
2/31/92/91/34/95/9
5/65/365/185/125/925/36
```

Глава 7
Массивы и строки

- ✦ Основы массивов
- ✦ Массивы как класс данных
- ✦ Массивы и объекты
- ✦ Строки
- ✦ Стандартный класс string в C++

В повседневной жизни мы обычно объединяем похожие объекты в группы. Мы покупаем горошек в банках и яйца в коробках. В языках программирования нам тоже необходимо группировать вместе данные одинакового типа. Основным механизмом, использующимся для этих целей в C++, является *массив*. Он может содержать от нескольких единиц данных до многих миллионов. Данные, сгруппированные в массиве, могут быть как основных типов, таких, как int или float, так и определенных пользователем типов, таких, как структуры или объекты.

Массивы похожи на структуры тем, что они тоже объединяют некоторое количество переменных в большой блок. Но в структуре обычно группируются переменные разных типов, а в массиве группируются однотипные данные. Более важное отличие в том, что к элементам структуры можно получить доступ по имени, а к элементам массива — по индексу. Использование индекса для каждого из элементов позволяет упростить доступ в случае большого количества элементов.

Массивы существуют почти в каждом языке программирования. Массивы языка C++ похожи на массивы других языков и идентичны массивам языка C.

В этой главе мы сначала рассмотрим массивы элементов основных типов данных, таких, как int и char. Затем мы рассмотрим массивы, использующиеся как члены классов, и массивы, содержащие в себе объекты. Таким образом, в этой главе мы планируем не только знакомство с массивами, но и расширение ваших знаний об ООП.

В стандарте C++ не только массивы предназначены для группирования элементов одного типа. Тип vector, находящийся в Стандартной библиотеке

шаблонов, является другим вариантом группировки. Мы рассмотрим векторы в главе 15 «Стандартная библиотека шаблонов (STL)».

В этой главе мы также рассмотрим два различных подхода к определению строк, которые используются для хранения текста и выполнения действий с ним. Первым видом строк является массив элементов типа char. Второй тип представляет собой объекты стандартного класса string в C++.

Основы массивов

Мы познакомимся с массивами на примере простой программы REPLAY. В ней создается массив, состоящий из четырех целых чисел, предназначенный для хранения данных о возрасте некоторых людей. Программа запрашивает ввод пользователем четырех значений, которые будут помещены в этот массив. В конце программы эти значения выводятся на дисплей.

```
// replay.cpp
// запоминание и вывод на экран информации, введенной пользователем
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  int age [ 4 ];
  for ( int j = 0; j < 4; j++ )
  {
    cout << "Введите возраст: ";
    cin age [ j ];
  }
  for ( j = 0; j < 4; j++ )
    cout << "Вы ввели: " << age [ j ] << endl;

  return 0;
}
```

Вот пример взаимодействия с программой:

```
Введите возраст: 44
Введите возраст: 16
Введите возраст: 23
Введите возраст: 68

Вы ввели: 44
Вы ввели: 16
Вы ввели: 23
Вы ввели: 68
```

В первом цикле for мы получаем значения возраста от пользователя и помещаем их в массив, затем, во втором цикле, мы считываем данные из массива и выводим их на дисплей.

Определение массивов

Как и другие переменные в C++, массив должен быть определен перед его использованием. Как и другие определения, определение массива включает в себя тип хранящихся в нем переменных и имя массива. Но помимо этого для массива необходимо указать размер, который определяет, сколько элементов массив может содержать. Размер следует за именем и заключается в квадратные скобки. На рис. 7.1 показан синтаксис определения массива.

В примере REPLAY использован массив типа int. За типом следует имя массива, затем открывается квадратная скобка, указывается размер массива и наконец скобка закрывается. Значение в скобках должно быть выражением, определяющим целую константу. В примере мы использовали значение 4.

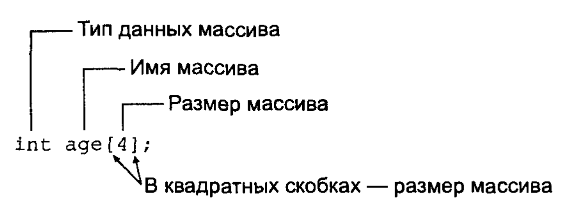


Рис. 7.1. Синтаксис определения массива

Элементы массива

Члены массива называют *элементами*. Как мы уже отмечали, все элементы массива одного типа, только значения у них разные. На рис. 7.2 показаны элементы массива age.

Согласно традиционному подходу, память на рисунке увеличивается вниз (хотя иногда изображают в обратном направлении). Поэтому первый элемент массива расположен в верхней части рисунка; следующие элементы располагаются ниже. Как записано в определении, массив может иметь только четыре элемента.

Заметим, что первый элемент массива имеет номер 0. Таким образом, так как всего в массиве четыре элемента, то последний элемент будет иметь номер 3. Эта ситуация иногда приводит к путанице: можно предположить, что последний элемент массива из четырех элементов должен иметь номер 4, но это не так.

Доступ к элементам массива

В примере REPLAY мы получаем доступ к элементам массива дважды. В первый раз — когда вставляем значение в массив в строке

```
cin >> age [ j ];
```

Во второй раз — когда мы считываем значения из массива в строке

```
cout << "\nВы ввели: << age [ j ];
```

В обоих случаях выражение для элемента массива будет таким:

```
age [ j ]
```

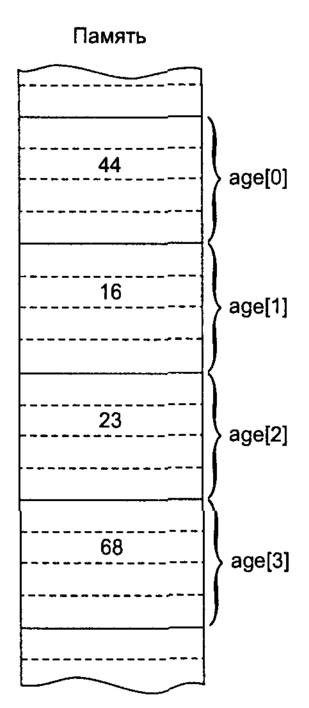


Рис. 7.2. Элементы массива

Оно состоит из имени массива и следующей за ним переменной j в квадратных скобках. Какой именно из элементов массива обозначает это выражение, зависит от значения j; age[0] ссылается на первый элемент, age[1] на второй, age[2] на третий, а age[3] на четвертый. Выражение, указывающееся в скобках, называется *индексом массива.*

Здесь j — переменная цикла для обоих циклов for, она увеличивается и принимает значения начиная с 0 и до 3; таким образом, мы последовательно получаем доступ к каждому из элементов массива.

Среднее арифметическое элементов массива

Вот другой пример работы массивов. В программе SALES пользователю предлагается ввести серию из шести значений, представляющих собой объемы продаж изделий за каждый день недели (исключая воскресенье), а затем программа вычисляет среднее арифметическое этих значений. Мы используем массив типа double, чтобы можно было пользоваться денежными значениями.

```
// sales.cpp
// определение среднего дневного объема продаж
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
int main ( )
```

```
{
  const int SIZE = 6;
  double sales [ SIZE ];

  cout << "Введите объем продаж на каждый из шести дней\n";
  for ( int j = 0; j < SIZE; j++ )
    cin >> sales [ j ];
  double total = 0;
  for ( j = 0; j < SIZE; j++ )
    total += sales [ j ];
  double average = tital / SIZE;
  cout << "Средний объем: " << average << endl;

  return 0;
}
```

Вот небольшой пример взаимодействия с программой SALES:

```
Введите объем продаж на каждый из шести дней
352.64
867.70
781.32
867.35
746.21
189.45
Средний объем: 643.11
```

Новая деталь в этой программе — это использование переменной константы для размера массива и в ограничениях цикла. Она определена в начале листинга:

```
const int SIZE = 6;
```

Использование переменной (вместо числа, такого, как 4, использованного в предыдущей программе) делает более простым изменение размера массива: нужно будет модифицировать только одну строку программы, чтобы изменился размер массива, ограничения циклов и значение в других местах, где встречается размер массива. Имя, написанное нами большими буквами, напоминает нам, что эта переменная не может быть изменена в программе.

Инициализация массива

Когда массив определен, мы можем присвоить его элементам значения. В примере DAYS 12 элементам массива days_per_month присваивается количество дней для каждого месяца.

```
// days.cpp
// показ количества дней с начала года и до введенной даты
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  int month, day, total_days;
  int days_per_month [ 12 ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
  cout << "\nВведите месяц (от 1 до 12): ";
  cin >> month;
  cout << "\nВведите день (от 1 до 31): ";
  cin >> day;
  total_days = day;
  for ( int j = 0; j < month - 1; j++ )
    total_days += days_per_month [ j ];
  cout << "Общее число дней с начала года: " << total_days << endl;

  return 0;
}
```

Программа вычисляет количество дней от начала года до даты, определенной пользователем. (Внимание: программа не работает для високосных лет.) Вот пример взаимодействия с программой:

```
Введите месяц (от 1 до 12): 3
Введите день (от 1 до 31): 11
Общее число дней с начала года: 70
```

Получив значения дня и месяца, программа присваивает значение дня переменной total_days. Затем она проходит через цикл, в котором к переменной прибавляются значения элементов массива days_per_month. Количество этих значений на единицу меньше, чем значение месяца. Например, если пользователь ввел пятый месяц, то значения первых четырех элементов массива (31, 28, 31 и 30) будут прибавлены к переменной total_days.

Значения, которыми инициализируется массив days_per_month, заключаются в скобки и разделяются запятыми. Они связаны с определением массива знаком равенства. На рис. 7.3 показан синтаксис инициализации массива.

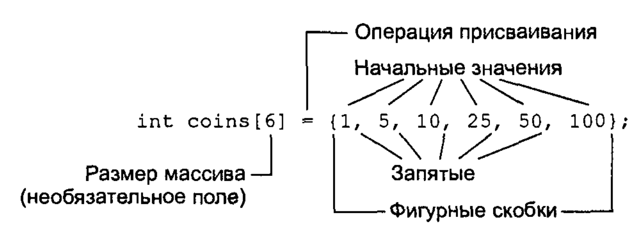


Рис. 7.3. Синтаксис инициализации массива

На самом деле нам не нужно использовать размер массива, когда мы инициализируем все элементы массива, так как компилятор может его вычислить, подсчитав инициализированные переменные. Таким образом, мы можем записать

```
int days_per_month [ 12 ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Что же случится, если вы явно использовали указание размера массива, но он не соответствует количеству элементов массива, которое инициализировано? Если инициализированы только несколько элементов, то оставшимся элементам будет присвоено значение 0. Если же инициализаций будет слишком много, то компилятор выдаст ошибку.

Многомерные массивы

До сих пор мы рассматривали одномерные массивы: одна переменная определяет каждый элемент массива. Но массивы могут иметь большую размерность. Вот программа SALEMON, в которой использован двумерный массив для хранения данных о продажах для нескольких отделов за несколько месяцев:

```
// salemon.cpp
// показ графика продаж
#include <iostream>
#include <iomanip>
using namespace std;

const int DISTRICTS = 4;
const int MONTHS = 3;
///////////////////////////////////////////////////////////
int main ( )
{
  int d, m;
  double sales [ DISTRICTS ][ MONTHS ];

  cout << endl;
  for ( d = 0; d < DISTRICTS; d++ )
    for ( m = 0; m < MONTHS; m++ )
    {
      cout << "Введите объем продаж для отдела " << d + 1;
      cout << ", за месяц " << m + 1 << ": ";
      cin >> sales [ d ][ m ];
    }

  cout << "\n\n";
  cout << "                        Месяц\n";
  cout << "                1         2         3";
  for ( d = 0; d < DISTRICTS; d++ )
  {
    cout << "\nОтдел " << d + 1;
    for ( m = 0; m < MONTHS; m++ )
      cout << setiosflags ( ios::fixed )
           << setiosflags ( ios::showpoint )
           << setprecision ( 2 )
           << setw ( 10 )
           << sales [ d ][ m ];
  }
  cout << endl;

  return 0;
}
```

Эта программа принимает от пользователя данные о продажах и затем выводит их в виде таблицы.

```
Введите объем продаж для отдела 1, за месяц 1: 3964.23
Введите объем продаж для отдела 1, за месяц 2: 4135.87
Введите объем продаж для отдела 1, за месяц 3: 4397.98
Введите объем продаж для отдела 2, за месяц 1: 867.75
```

```
Введите обьем продаж для отдела 2, за месяц 2: 923.59
Введите обьем продаж для отдела 2, за месяц 3: 1037.01
Введите обьем продаж для отдела 3, за месяц 1: 12.77
Введите обьем продаж для отдела 3, за месяц 2: 378.32
Введите обьем продаж для отдела 3, за месяц 3: 798.22
Введите обьем продаж для отдела 4, за месяц 1: 2983.53
Введите обьем продаж для отдела 4, за месяц 2: 3983.73
Введите обьем продаж для отдела 4, за месяц 3: 9494.98

Месяц
123
Отдел 13964.234135.B74397.98
Отдел 2867.75923.591037.01
Отдел 312.77378.32798.22
Отдел 42983.533983.739494.98
```

Определение многомерного массива

Массив определяется двумя размерами, которые заключены в квадратные скобки:

```
double sales [ DISTRICTS ][ MONTHS ];
```

Вы можете считать sales двумерным массивом, похожим на шахматную доску. Иными словами можно сказать, что этот массив является массивом массивов. Это массив элементов DISTRICTS, каждый из которых является массивом элементов MONTHS. На рис. 7.4 показано, как это выглядит.

Конечно, массивы могут иметь большую размерность. Трехмерный массив — это массив массивов, которые состоят из массивов. Доступ к элементам массива осуществляется с использованием трех индексов:

```
elem = dimen3 [ x ][ y ][ z ];
```

Аналогично обстоит дело с одно- и двумерными массивами.

Доступ к элементам многомерного массива

Элементы двумерного массива требуют двух индексов:

```
sales [ d ][ m ]
```

Заметим, что каждый индекс заключается в отдельные квадратные скобки. Запятые не используются. Нельзя писать sales[d, m]; это работает в некоторых языках, но не в C++.

Форматирование чисел

Программа SALEMON выводит таблицу денежных значений. Важно, чтобы такие значения были правильно отформатированы. Давайте отступим от нашей темы и рассмотрим, как это делается в C++. Для денежных значений обычно используют два знака после запятой. Мы хотим выровнять значения в столбце. Также хорошо бы выводить последний ноль, то есть 79.50, а не 79.5.

Убедимся, что не потребуется много работы, чтобы реализовать это все с помощью потоков ввода/вывода C++. Вы уже видели, как использовался метод setw() для установки ширины выводимого поля. Форматирование десятичных чисел требует нескольких добавочных методов.

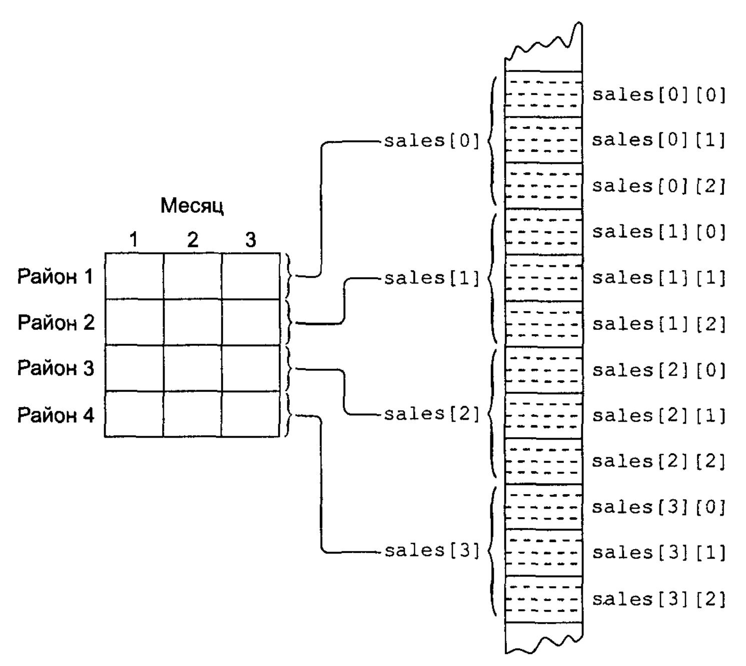


Рис. 7.4. Двумерный массив

Вот строка, которая печатает числа с плавающей точкой, именованные как fpn, в поле шириной в 10 символов с двумя знаками после запятой:

```
cout << setiosflags ( ios::fixed )      // нормальный неэкспоненциальный вид
     << setiosflags ( ios::showpoint ) // всегда показывать десятичную точку
     << setprecision ( 2 )              // два знака после запятой
     << setw ( 10 )                     // ширина вывода в 10 символов
     << fpn;                            // само число
```

Группа битовых форматирующих флагов в long int класса ios определяет, как должно быть выполнено форматирование. При этом нам не нужно знать, что такое класс ios или каковы причины использования существующего синтаксиса этого класса для осуществления работы выражений.

Мы коснемся только двух флагов: ios::fixed и ios::showpoint. Для их установки используйте метод setiosflags с именем флага в качестве аргумента. Имя должно предшествовать имени класса ios и отделяться от него операцией разрешения (::).

В первых двух строках оператора cout устанавливаются флаги ios (если вам нужно убрать флаги в более поздней точке программы, вы можете использовать метод resetiosflags). Флаг fixed предотвращает печать числа в экспоненциальной форме, например 3.45e3. Флаг showpoint определяет положение десятичной точки, даже если число не имеет дробной части: 123.00 вместо 123.

Для установки точности до двух знаков после запятой используйте метод setprecision с числом в качестве аргумента. Мы уже знаем, как установить ширину поля, используя метод setw. При установке всех этих параметров для cout вы сможете получить желаемый формат вывода числа.

Подробнее мы поговорим о флагах форматирования ios в главе 12 «Потоки и файлы».

Инициализация многомерных массивов

Как вы могли ожидать, вы можете инициализировать многомерные массивы. Только предварительно подготовьтесь к тому, что придется напечатать много скобок и запятых. Вот вариант программы SALEMON, которая использует инициализацию массива вместо запроса пользователя о вводе. Эта программа называется SALEINIT.

```
// saleinit.cpp
// показ графика продаж по данным массива
#include <iostream>
#include <iomanip>
using namespace std;
const int DISTRICTS = 4; // размеры массива
const int MONTHS = 3;
///////////////////////////////////////////////////////////
int main ( )
{
  int d, m;

  double sales [ DISTRICTS ][ MONTHS ] =
  {
    {  1432.07,   234.50,   654.01 },
    {   322.00, 13838.32, 17589.88 },
    {  9328.34,   934.00,  4492.30 },
    { 12838.29,  2332.63,    32.93 }
  };
  cout << "\n\n";
  cout << "                        Месяц\n";
  cout << "                 1         2         3";
  for ( d = 0; d < DISTRICTS; d++ )
  {
    cout << "\nОтдел " << d + 1;
    for ( m = 0; m < MONTHS; m++ )
    {
      cout << setw ( 10 ) << setiosflags ( ios::fixed )
           << setiosflags ( ios::showpoint ) << setprecision ( 2 )
           << sales [ d ] [ m ];
    }
  }
  cout << endl;
  return 0;
}
```

Напомним, что двумерный массив в действительности является массивом массивов. Формат инициализации такого массива базируется на этом факте. Инициализирующие значения для каждого подмассива заключены в скобки и разделены запятыми:

```
{ 1432.07, 234.50, 654.01 },
```

а затем все эти подмассивы, каждый из которых является элементом главного массива, также заключаются в скобки и разделяются запятыми, как мы видели в листинге.

Передача массивов в функции

Массивы могут быть использованы как аргументы функций. Примером может служить вариант программы SALEINIT, в котором массивы объемов продаж передаются в функцию, выводящую данные в виде таблицы. Взгляните на листинг программы SALEFUNC[1]:

```
// salefunc.cpp
// передача массива в виде параметра
#include <iostream>
#include <iomanip>
using namespace std;
const int DISTRICTS = 4;
const int MONTHS = 3;
void display ( double [ DISTRICTS ][ MONTHS ] );
////////////////////////////////////////////////////////////
int main ( )
{
  double sales [ DISTRICTS ][ MONTHS ] =
  {
    {  1432.07,   234.50,   654.01 },
    {   322.00, 13838.32, 17589.88 },
    {  9328.34,   934.00,  4492.30 },
    { 12838.29,  2332.63,    32.93 }
  };

  display ( sales );
  cout << endl;
return 0;
}
////////////////////////////////////////////////////////////
// display ( )
// функция для вывода на экран массива
void display ( double funsales [ DISTRICTS ][ MONTHS ] )
{
  int d, m;

  cout << "\n\n";
  cout << "                        Месяц\n";
  cout << "                1         2         3";

  for ( d = 0; d < DISTRICTS; d++ )
  {
    cout << "\nОтдел " << d + 1;
    for ( m = 0; m < MONTHS; m++ )
    {
      cout << setiosflags ( ios::fixed ) << setw ( 10 )
           << setiosflags ( ios::showpoint ) << setprecision ( 2 )
           << funsales [ d ] [ m ];
    }
  }
}
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

Объявление функции с аргументами в виде массивов

В объявлениях функции массивы-аргументы представлены типом данных и размером. Вот объявление функции display():

```
void display ( float [ DISTRICTS ][ MONTHS ] );
```

На самом деле здесь есть один необязательный элемент. Следующее объявление работает так же:

```
void display ( float [ ][ MONTHS ] );
```

Почему функции не нужно значение первой размерности? Вспомним, что двумерный массив — это массив массивов. Функция сначала рассматривает аргумент как массив отделов. Ей не важно знать, сколько отделов, но нужно знать, насколько велики элементы, представляющие собой отделы, так как тогда она сможет вычислить, где находится каждый элемент (умножая количество байтов, приходящееся на один элемент, на индекс нужного элемента). Поэтому мы можем сообщить функции размер массива MONTHS, но не сообщать, сколько таких массивов находится в массиве DISTRICTS.

Отсюда следует, что если мы объявили функцию с одномерным массивом в качестве аргумента, то нам не нужно указывать размер массива:

```
void somefunc ( int elem [ ] );
```

Вызов функции с массивом в качестве аргумента

При вызове функции в качестве аргумента используется только имя массива.

```
display ( sales );
```

Это имя (в нашем случае sales) в действительности представляет собой адрес массива в памяти. Мы не будем объяснять адреса детально до главы 10 «Указатели», но рассмотрим несколько общих вопросов.

Использование адресов для массивов-аргументов похоже на использование аргумента ссылки, при котором значения элементов массива не дублируются (копируются) в функции. (Смотрите обсуждение ссылочных аргументов в главе 5 «Функции».) Вместо этого функция работает с оригинальным массивом, хотя ссылается на него, используя другое имя. Эта система используется для массивов, потому что они могут быть очень большими; дублирование массива для каждой вызывающей его функции может отнимать много времени и пространства в памяти.

Однако адрес — это не то же самое, что и ссылка. С именем массива не используется амперсанд (&) при объявлении функции. Пока мы не изучили указатели, примите на веру, что массивы передаются в функцию только с использованием их имени и что функция работает при этом с оригиналом массива, а не с его копией.

Определение функции с массивом в качестве аргумента

Объявление функции выглядит следующим образом:

```
void display ( double funsales [ DISTRICTS ][ MONTHS ] );
```

При записи аргумента-массива используются тип его данных, имя массива и его размерности. Имя массива, используемое при объявлении функции (в нашем случае funsales), может отличаться от имени массива, который затем будет использоваться в функции (sales), но оба этих имени ссылаются на один и тот же массив. Должны быть определены все размерности массива (исключая первую в некоторых случаях); они нужны функции для правильного обращения к элементам массива.

Обращаясь к элементам массива, функция использует то имя массива, которое было использовано при ее объявлении:

```
funsales [ d ][ m ]:
```

При всех остальных операциях функции с массивом она действует так, как если бы массив был определен в самой функции.

Массивы структур

Массивы могут содержать в себе не только данные основных типов, но и структуры. Вот пример, основанный на структуре part из главы 4 «Структуры».

```
// partaray.cpp
// массив из структур
#include <iostream>
using namespace std:
const int SIZE = 4:
/////////////////////////////////////////////////////////////
struct part
{
  int modelnumber:
  int partnumber:
  float cost:
}:
/////////////////////////////////////////////////////////////
int main ( )
{
  int n:
  part apart [ SIZE ]:

  for ( n = 0: n < SIZE: n++ )
  {
    cout << endl:
    cout << "Введите номер модели: ":
    cin >> apart [ n ].modelnumber:
    cout << "Введите номер части: ":
    cin >> apart [ n ].partnumber:
    cout << "Введите стоимость: ":
    cin >> apart [ n ].cost:
  }
  cout << endl:
  for ( n = 0: n < SIZE: n++ )
  {
    cout << "Модель " << apart [ n ].modelnumber:
```

```
    cout << "  Часть  " << apart [ n ].partnumber;
    cout << "  Стоимость  " << apart [ n ].cost << endl;
  }
  return 0;
}
```

Пользователь вводит номер модели, номер части и стоимость части. Программа записывает эти данные в структуру. Однако эта структура является только одним из элементов массива структур. Программа запрашивает данные для разных частей и хранит их в четырех элементах массива apart. Затем она выводит информацию. Примеры вывода программы:

```
Введите номер модели: 44
Введите номер части: 4954
Введите стоимость: 133.45

Введите номер модели: 44
Введите номер части: B431
Введите стоимость: 97.59

Введите номер модели: 77
Введите номер части: 9343
Введите стоимость: 109.99

Введите номер модели: 77
Введите номер части: 4297
Введите стоимость: 3456.55

Модель 44 Часть 4954 Стоимость 133.45
Модель 44 Часть 8431 Стоимость 97.59
Модель 77 Часть 9343 Стоимость 109.99
Модель 77 Часть 4297 Стоимость 3456.55
```

Массив структур определен в строке:

```
part apart [ SIZE ];
```

Здесь применен тот же синтаксис, что и для массивов, использующих основные типы данных. Только имя типа part показывает нам, что этот массив содержит данные более сложного типа.

Доступ к данным, членам структуры, которая является элементом массива, требует нового синтаксиса. Например,

```
apart [ n ].modelnumber
```

ссылается на переменную modelnumber структуры, которая является n элементом массива apart. На рис. 7.5 показано, как это выглядит.

Массивы структур — это полезный тип данных, используемый в различных ситуациях. Мы показали массив запчастей для машины, но мы могли хранить в массиве и личные данные сотрудников (имя, возраст, зарплата), и географические особенности городов (название, количество населения, высота над уровнем моря), и многое другое.

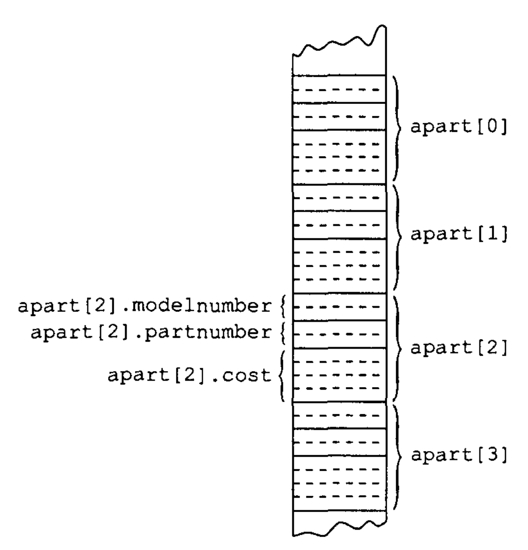


Рис. 7.5. Массив структур

Массивы как члены классов

Массивы могут быть использованы в качестве полей класса. Давайте рассмотрим пример, который моделирует компьютерную структуру данных — стек.

Стек работает как пружинное устройство, удерживающее патроны в магазине. Когда вы помещаете патрон в магазин, стек немного оседает; когда вы вынимаете патрон, то он поднимается. Последний положенный в магазин патрон всегда будет вынут первым.

Стек — это один из краеугольных камней архитектуры микропроцессоров, используемой в самых современных компьютерах. Как мы упоминали ранее, функции передают аргументы и хранят возвращаемые адреса в стеке. Этот вид стека частично реализован в оборудовании, и к нему удобнее всего получать доступ через язык ассемблера. Однако стек можно полностью реализовать в программном обеспечении. Здесь стек представляет собой полезное устройство для хранения данных в определенной программной ситуации, например такой, как разбор алгебраического выражения.

В нашем примере, программе STAKARAY, создадим простой класс stack.

```
// stakaray.cpp
// класс стек
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Stack
{
  private:
    enum { MAX = 10 }; // немного нестандартный синтаксис
    int st [ MAX ];    // стек в виде массива
    int top;           // вершина стека
  public:
```

```
    Stack ( ): // конструктор
      { top = 0; }
    void push ( int var ) // поместить в стек
      { st [ ++top ] = var; }
    int pop ( ) // взять из стека
      { return st [ top-- ]; }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Stack s1;

  s1.push ( 11 );
  s1.push ( 22 );
  cout << "1: " << s1.pop ( ) << endl;
  cout << "2: " << s1.pop ( ) << endl;
  s1.push ( 33 );
  s1.push ( 44 );
  s1.push ( 55 );
  s1.push ( 66 );
  cout << "3: " << s1.pop ( ) << endl;
  cout << "4: " << s1.pop ( ) << endl;
  cout << "5: " << s1.pop ( ) << endl;
  cout << "6: " << s1.pop ( ) << endl;
  return 0;
}
```

Важным членом класса stack является массив st. Переменная top типа int хранит в себе индекс последнего элемента, положенного в стек; этот элемент располагается на вершине стека.

Размер массива, используемого для стека, определяется переменной MAX в строке

```
enum { MAX = 10 };
```

Это определение переменной MAX необычно. Придерживаясь философии инкапсуляции, предпочтительнее определять константы, используемые в классе, внутри класса — здесь так определена переменная MAX. Таким образом, использование глобальных констант-переменных для таких целей нецелесообразно. Стандарт C++ дает возможность объявления константы MAX внутри класса:

```
static const int MAX = 10;
```

Это означает, что MAX — константа и применяется для всех объектов класса. К сожалению, некоторые компиляторы не позволяют использовать эту конструкцию.

Используя обходной путь, мы можем определить такие константы через перечисление (это описано в главе 4). Нам не нужно именовать само перечисление, нам нужен только один его элемент:

```
enum { MAX = 10 };
```

Здесь определена переменная MAX как целое со значением 10, определение дано внутри класса. Этот подход работает, но он неудобен. Если ваш компиля-

тор поддерживает подход static const, то вам следует использовать его вместо определения констант внутри класса.

На рис. 7.6 показан стек. Так как на рисунке значения памяти возрастают по направлению сверху низ, то вершина стека находится в нижней части рисунка. При добавлении в стек нового элемента индекс переменной top увеличивается, и она будет показывать на новую вершину стека. При удалении элемента из стека индекс переменной top уменьшается. (Нам не нужно стирать старое значение из памяти при удалении элемента, оно просто становится несущественным.)

Для помещения элемента в стек — процесс называемый проталкиванием — вы вызываете метод push() с сохраняемым значением в качестве аргумента. Для извлечения элемента из стека вы вызываете метод pop(), который возвращает значение элемента.

Программа main() примера STAKARAY применяет класс stack, создавая объект s1 этого класса. Программа помещает два элемента в стек, затем извлекает их и выводит на дисплей. Затем в стек помещаются еще четыре элемента, далее они извлекаются и выводятся. Вот вывод программы:

```
1: 22
2: 11
3: 66
4: 55
5: 44
6: 33
```

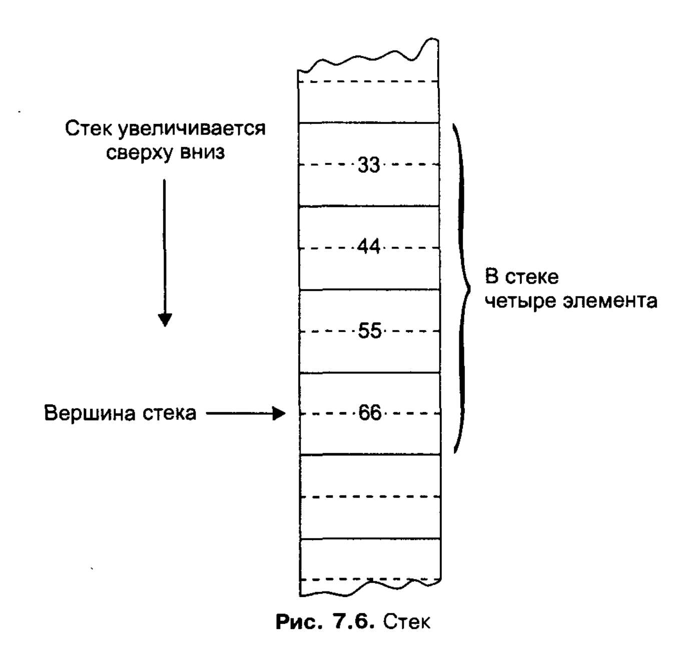


Рис. 7.6. Стек

Как вы можете видеть, элементы извлекаются из стека в обратном порядке; последний элемент, помещенный в стек, будет первым элементом, извлеченным оттуда.

Заметим небольшое отличие в использовании префиксной и постфиксной формы записи операторов увеличения и уменьшения. В строке

```
st [ ++top ] = var;
```

метода push() сначала переменная top увеличивается и будет показывать уже на следующий элемент массива, расположенный за текущим последним элементом. Затем этому элементу присваивается значение переменной var, и он становится новой вершиной стека. В строке

```
return st [ top-- ];
```

сначала возвращается значение вершины стека, а затем уменьшается переменная top, так, что она будет уже указывать на предыдущий элемент.

Класс stack является примером важной возможности ООП; он используется для реализации контейнера или механизма для хранения данных. В главе 15 мы увидим, что стек может быть использован не только для этой цели. Он применяется в связных списках, очередях, группах и т. д. Механизм хранения данных выбирается в соответствии с требованиями программы. Использование существующих классов для хранения данных означает, что программисту не нужно будет тратить время на их дублирование.

Массивы объектов

Мы видели, что объекты могут содержать массивы. Но верно и обратное: мы можем создать массив объектов. Рассмотрим две ситуации: массив интервалов и колоду карт.

Массивы интервалов

В главе 6 «Объекты и классы» мы показали несколько примеров класса Distance, который объединяет футы и дюймы в своем объекте, представляющем собой новый тип данных. Наша следующая программа ENGLARAY демонстрирует массив таких объектов.

```
// englaray.cpp
// объекты для английских мер
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class Distance
{
  private:
    int feet;
    float inches;
  public:
    void getdist ( ) // получение информации
      {
        cout << "\n  Введите футы: "; cin >> feet;
        cout << "  Введите дюймы: "; cin >> inches;
```

```
    }
    void showdist ( ) const // показ информации
      { cout << feet << "\'-" << inches '\"'; }
};
//////////////////////////////////////////////////////////
int main ( )
{
  Distance dist [ 100 ]; // массив переменных
  int n = 0;             // счетчик данных
  char ans;              // ответ пользователя (д/н)

  cout << endl;

  do
  {
    cout << "Введите длину номер " << n + 1;
    dist [ n++ ].getdist ( );      // получаем и сохраняем длину
    cout << "Продолжить ввод (д/н)?: ";
    cin >> ans;
  }
  while ( ans != 'н' );           // продолжать, пока не будет введено 'н'

  // показываем все введенное
  for ( int j = 0; j < n; j++ )
  {
    cout << "\nДлина номер " << j + 1 << " : ";
    dist [ j ].showdist ( );
  }

  cout << endl;

  return 0;
}
```

В этой программе пользователь может ввести произвольное количество интервалов. После введения каждого интервала программа спрашивает пользователя о том, нужно ли будет ввести еще один интервал. Если нет, то программа выводит все введенные ранее интервалы. Приведем пример взаимодействия с программой; здесь пользователь вводит три интервала:

```
Введите длину номер 1
  Введите футы: 5
  Введите дюймы: 4
Продолжить ввод (д/н)?: д
Введите длину номер 2
  Введите футы: 6
  Введите дюймы: 2.5
Продолжить ввод (д/н)?: д
Введите длину номер 3
  Введите футы: 5
  Введите дюймы: 10.75
Продолжить ввод (д/н)?: н

Длина номер 1 : 5'-4"
Длина номер 2 : 6'-2.5"
Длина номер 3 : 5'-10.75"
```

Конечно же, вместо простого вывода интервалов на дисплей программа может подсчитать их среднее значение, записать их на диск или выполнить какие-либо другие операции.

Границы массива

В этой программе ввод данных пользователем организован в виде цикла. Здесь пользователь может ввести сколько угодно структур типа part, даже больше значения MAX (равного здесь 100), представляющего собой размер массива.

Что случится, если пользователь введет более 100 интервалов? Что-то непредвиденное, но почти определенно плохое. В C++ нет проверки границ массива. Если программа поместит что-то за пределами массива, то компилятор и исполняемая программа протестовать не будут. Однако эти данные могут быть записаны поверх других данных или поверх самой программы. Это может послужить причиной странных эффектов или даже полного краха системы.

Мораль всего этого такова, что программисту следует применять проверку границ массива. Если возникает возможность переполнения массива данными, то он должен иметь большие размеры или у него должна быть возможность предупредить пользователя о возможном переполнении. Например, вы можете вставить этот код в начало цикла программы ENGLARAY:

```
if ( n >= MAX )
{
  cout << "\nМассив полон!!!";
  break;
}
```

Таким образом вы сможете прервать цикл и предотвратить переполнение массива.

Доступ к объектам в массиве

Объявление класса Distance в этой программе похоже на то, которое использовалось в предыдущей программе. Однако в функции main() мы определили массив объектов класса Distance:

```
Distance dist [ MAX ];
```

Здесь типом данных массива dist является класс Distance. Число элементов массива определяется значением переменной MAX. На рис. 7.7 показано, как это выглядит.

Доступ к методам объектов элементов массива похож на доступ к членам структуры, являющейся элементом массива, как в примере PARTARAY. Здесь показано, как вызвать метод showdist() элемента массива dist под номером j:

```
dist [ j ].showdist ( );
```

Как вы можете видеть, доступ к методу объекта, являющегося элементом массива, осуществляется с помощью операции точки. За именем массива следу-

ют квадратные скобки, в которые заключен индекс элемента, затем операция точки и имя метода, за ними следуют скобки. Это похоже на доступ к полю структуры (или класса), за исключением того, что вместо имени переменной используется имя метода со скобками.

Заметим, что при вызове метода getdist() для помещения интервала в массив мы воспользуемся случаем и увеличим индекс массива n:

```
dist [ n++ ].getdist ( );
```

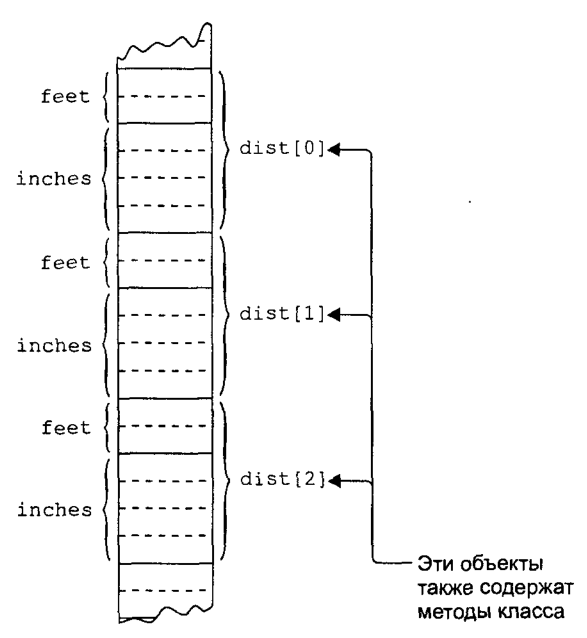


Рис. 7.7. Массив объектов

Таким образом, следующая группа данных, полученная от пользователя, будет помещена в структуру следующего элемента массива dist. Переменная n должна быть увеличена вручную, как здесь, потому что мы используем цикл do вместо цикла for. В цикле for переменная цикла, которая увеличивается автоматически, может служить индексом массива.

Массивы карт

Вот другой, довольно длинный, пример массива объектов. Вы, конечно, помните пример CARDOBJ из главы 6. Мы воспользуемся классом card из этого примера и определим массив из 52 объектов этого класса, создав таким образом колоду карт. Вот листинг программы CARDARAY:

```
// cardaray.cpp
// класс игральных карт
#include <iostream>
#include <cstdlib>
```

```
#include <ctime>
using namespace std:

enum Suit { clubs. diamonds. hearts. spades }:
// от 2 до 10 обычные числа
const int jack = 11:
const int queen = 12;
const int king = 13:
const int ace = 14:
//////////////////////////////////////////////////////////////
class card
{
  private:
    int number:
    Suit suit:
  public:
    card ( ) // конструктор
      { }
    void set ( int n. Suit s ) // установка значения
      { suit = s: number = n: }
    void display ( ):          // показ карты
}:
//////////////////////////////////////////////////////////////
void card::display ( )
{
  if ( number >= 2 && number <= 10 )
    cout << number;
  else
  {
    switch ( number )
    {
      case jack: cout << 'J': break:
      case queen: cout << 'Q': break;
      case king: cout << 'K': break:
      case ace: cout << 'A': break:
    }
  }
  switch ( suit )
  {
    case clubs: cout << static_cast<char>( 5 ): break:
    case diamonds: cout << static_cast<char>( 4 ): break;
    case hearts: cout << static_cast<char>( 3 ): break;
    case spades: cout << static_cast<char>( 6 ): break;
  }
}
//////////////////////////////////////////////////////////////
int main ( )
{
  card deck [ 52 ]:
  int j:

  cout << endl:
  for ( j = 0: j < 52: j++ )   // создаем упорядоченную колоду карт
  {
    int num = ( j % 13 ) + 2:
    Suit su = Suit ( j / 13 ):
    deck [ j ].set ( num. su ):
  }
```

```
  // показываем исходную колоду
  cout << "Исходная колода:\n";
  for ( j = 0; j < 52; j++ )
  {
    deck [ j ].display ( );
    cout < "  ";
    if ( !( ( j + 1 ) % 13 ) ) // начинаем новую строку после каждой 13-й карты
      cout << endl;
  }
  srand ( time ( NULL ) );     // инициализируем генератор случайных чисел
  for ( j = 0; j < 52; j++ )
  {
    int k = rand ( ) % 52;     // выбираем случайную карту
    card temp = deck [ j ];    // и меняем ее с текущей
    deck [ j ] = deck [ k ];
    deck [ k ] = temp;
  }
  // показываем перемешанную колоду
  cout << "\nПеремешанная колода:\n";
  for ( j = 0; j < 52; j++ )
  {
    deck [ j ].display ( );
    cout << "  ";
    if ( !( ( j + 1 ) % 13 ) ) // начинаем новую строку после каждой 13-й карты
      cout << endl;
  }
  return 0;
}
```

Теперь, раз уж мы создали колоду карт, нам, наверное, стоит перемешать их. В программе мы показали карты, перемешали их и показали вновь. Для экономии пространства мы использовали графические символы мастей: трефы, черви, пики и бубны. На рис. 7.8 показан вывод программы. В эту программу включены несколько новых идей, давайте рассмотрим их по очереди.

```
Упорядоченная колода
2♣ 3♣ 4♣ 5♣ 6♣ 7♣ 8♣ 9♣ 10♣ J♣ Q♣ K♣ A♣
2♦ 3♦ 4♦ 5♦ 6♦ 7♦ 8♦ 9♦ 10♦ J♦ Q♦ K♦ A♦
2♥ 3♥ 4♥ 5♥ 6♥ 7♥ 8♥ 9♥ 10♥ J♥ Q♥ K♥ A♥
2♠ 3♠ 4♠ 5♠ 6♠ 7♠ 8♠ 9♠ 10♠ J♠ Q♠ K♠ A♠

Смешанная колода
3♠ 9♦ 6♦ K♠ 8♥ 4♠ 7♦ 4♣ 3♦ 3♣ A♥ 2♦ 9♣
6♣ 7♣ 9♥ 8♠ Q♣ Q♦ 10♥ J♣ 6♥ 4♥ J♦ K♣ 5♠
3♥ J♠ 5♣ K♦ Q♥ 10♦ 8♦ 2♣ 6♠ A♠ 4♦ J♥ 8♣
10♠ 2♥ Q♠ 10♣ 5♦ A♣ K♥ 7♥ 5♥ A♦ 2♠ 9♠ 7♠
```

Рис. 7.8. Вывод программы CARDARAY

Графические символы

Здесь мы использовали несколько специальных графических символов из кодов ASCII (см. приложение А «Таблица ASCII», список кодов ASCII.) В методе display() класса card мы использовали коды 5, 4, 3 и 6 для получения символов

треф, бубен, червей и пик соответственно. Преобразуя эти номера к типу char, как в следующей строке,

```
static_cast<char>( 5 )
```

мы сможем напечатать их в виде символов, а не номеров.

Колода карт

Массив структур, предназначенный для колоды карт, определен в строке

```
card deck [ 52 ];
```

Здесь создается массив, названный deck, содержащий 52 объекта типа card. Для вывода j карты колоды мы вызовем метод display():

```
deck [ j ].display ( );
```

Случайные номера

Всегда весело, но иногда даже полезно генерировать случайные числа. В этой программе мы используем их для перемешивания колоды. Для получения случайных чисел необходимы два шага. Сначала генератор случайных чисел должен быть инициализирован, то есть должно быть задано начальное число. Для этого вызываем библиотечную функцию srand (). Эта функция использует в качестве начального числа системное время. Она требует двух заголовочных файлов: CSTDLIB и CTIME.

Здесь для генерации случайного числа мы вызвали библиотечную функцию rand( ). Она возвращает случайное целое число. Для получения числа из диапазона от 0 до 51 мы применили операцию получения остатка от деления на 52 к результату функции rand().

```
int k = rand ( ) % 52;
```

Получившееся случайное число k затем используется как индекс для обмена двух карт. Мы выполняем цикл for, меняя местами карту, чей индекс принадлежит картам от 0 до 51, с другой картой, чей индекс является случайным числом. Когда все 52 карты поменяются местами со случайной картой, колода будет считаться перемешанной. Эта программа может считаться основой для программ, реализующих карточные игры, но мы оставим детали для вас.

Массивы объектов широко используются в программировании на C++. Мы рассмотрим и другие примеры далее.

Строки

В начале этой главы мы заметили, что обычно в C++ используются два вида строк: строка как массив символов типа char и строка как объект класса string. В этом разделе мы опишем первый тип строк, который относится к нашей теме, так как строка — это массив элементов типа char. Мы называем его строковым, так как ранее он был единственным типом представления строк в C (и ранних

версиях C++). Этот тип можно также назвать char*-строками, так как он может быть представлен в виде указателя на char. (* означает указатель, мы изучим это в главе 10.)

Тем не менее строки, созданные с помощью класса string, который мы рассмотрим в следующем разделе, во многих ситуациях вытеснили строковый тип. Но строковый тип все еще используется по многим причинам. Во-первых, он используется во многих библиотечных функциях языка C. Во-вторых, он год за годом продолжает появляться в кодах. И в-третьих, для изучающих C++ строковый тип наиболее примитивен и поэтому легко понимается на начальном уровне.

Строковые переменные

Как и другие типы данных, строки могут быть переменными и константами. Мы рассмотрим эти две сущности перед тем, как начать рассматривать более сложные строковые операции. Вот пример, в котором определена одна строковая переменная. (В этом разделе мы будем предполагать, что слово строка означает строковый тип.) Программа просит пользователя ввести строку и помещает эту строку в строковую переменную, а затем выводит эту строку на дисплей. Листинг программы STRINGIN:

```
// stringin.cpp
// простая переменная строка
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  const int MAX = 80; // максимальный размер строки
  char str [ MAX ];   // сама строка

  cout << "Введите строку: ";          // ввод строки
  cin >> str;

  cout << "Вы ввели: " << str << endl; // показ результата

  return 0;
}
```

Определение строковой переменной str выглядит похожим (да так оно и есть) на определение массива типа char:

```
char str [ MAX ];
```

Мы используем операцию >> для считывания строки с клавиатуры и помещения ее в строковую переменную str. Этой операции известно, как работать со строками: она понимает, что это массив символов. Если пользователь введет строку «Секретарь» (тот, кто копирует рукописи) в этой программе, то массив str будет выглядеть чем-то похожим на рис. 7.9.

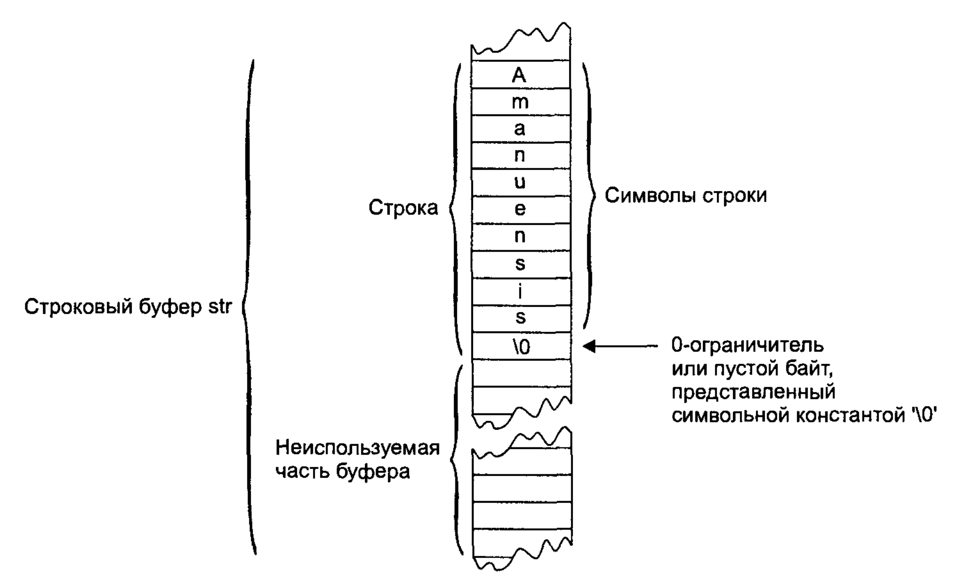


Рис. 7.9. Строка, хранящаяся в строковой переменной

Каждый символ занимает 1 байт памяти. Важная деталь, касающаяся строк, состоит в том, что они должны завершаться байтом, содержащим 0. Это часто представляют символьной константой \0, код которой в ASCII равен 0. Завершающий ноль называется *нулевым символом*. Когда операция << выводит строку, то она выводит символы до тех пор, пока не встретит нулевой символ.

Как избежать переполнения буфера

Программа STRINGIN предлагает пользователю напечатать строку. Что случится, если пользователь введет строку, которая окажется длиннее, чем массив, используемый для ее хранения? Как мы упомянули ранее, в C++ нет встроенного механизма, защищающего программу от помещения элементов за пределы массива. Поэтому слишком увлеченный пользователь может привести систему к краху.

Однако существует возможность ограничить при использовании операции >> количество символов, помещаемых в массив. Программа SAFETYIN демонстрирует этот подход.

```
// safetyin.cpp
// избежание переполнения буфера
#include <iostream>
#include <iomanip>
using namespace std:
////////////////////////////////////////////////////////////
int main ( )
{
  const int MAX =20:          // максимальный размер строки
  char str [ MAX ]:           // сама строка

  cout << "\nВведите строку: ":
```

```
  cin >> setw ( MAX ) >> str; // ввод не более чем MAX символов

  cout << "Вы ввели: " << str << endl;

  return 0;
}
```

Эта программа использует метод setw, определяющий максимальное количество символов, которое сможет принять буфер. Пользователь может напечатать больше символов, но операция >> не вставит их в массив. В действительности операция вставит в массив на один символ меньше, чем определено, так как в буфере есть место для завершающего нулевого символа. Таким образом, в программе SAFETYIN максимальным числом возможных символов будет 19.

Строковые константы

Вы можете инициализировать строку постоянным значением при ее определении. Вот пример STRINIT, в котором это сделано (будем надеяться, что Александр Сергеевич не обидится на такое использование его стихов):

```
// strinit.cpp
// инициализация строки
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  char str [ ] = "Я памятник себе воздвиг нерукотворный.";
  cout << str << endl;

  return 0;
}
```

Здесь строковая константа записана как нормальная фраза, ограниченная кавычками. Это может показаться необычным, так как строка — это массив типа char. В последнем примере мы видели, что массив инициализируется рядом значений, заключенных в скобки и разделенных запятыми. Почему переменная str не инициализирована так же? В действительности вы можете использовать такую последовательность символьных констант:

```
char str [ ] = { 'Я', ' ', 'п', 'а', 'м', 'я', 'т', 'н', 'и', 'к', ' ', 'с', 'е', 'б', 'е'
```

и т. д. К счастью, разработчики C++ (и C) сжалились над нами и предоставили нам упрощенный подход, показанный в программе STRINIT. Эффект тот же самый: символы помещаются один за другим в массив. Как во всех строках, последним символом будет нулевой.

Чтение внутренних пробелов

Если вы введете в программе STRINGIN строку, содержащую больше одного слова, то вы будете неприятно удивлены. Вот пример:

```
Введите строку: Идет бычок, качается, вздыхает на ходу.
Вы ввели: Идет
```

Куда же делся остаток фразы? Оказывается, что операция >> считает пробел нулевым символом. Таким образом, он считывает строки, состоящие из одного слова, и что-либо, напечатанное после пробела, отбрасывается.

Для считывания строк, содержащих пробелы, мы используем другой метод — cin.get(). Этот синтаксис означает использовать метод get() класса stream для его объекта cin. В следующем примере, BLANKSIN, показано, как это работает.

```
// blanksin.cpp
// ввод строки с пробелами
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
int main ( )
{
  const int MAX = 80; // максимальная длина строки
  char str [ MAX ];   // сама строка

  cout << "\nВведите строку: ";
  cin.get ( str, MAX );
  cout << "Вы ввели: " << str << endl;

  return 0;
}
```

Первый аргумент метода cin::get() — это адрес массива, куда будет помещена введенная строка. Второй аргумент определяет максимальный размер массива, автоматически предупреждая, таким образом, его переполнение.

При использовании этого метода введенная строка сохранится полностью.

```
Введите строку: Что посеешь, то и пожнешь.
Вы ввели: Что посеешь, то и пожнешь.
```

Проблемы могут возникнуть, если вы перепутаете метод cin.get() с операциями cin и >>. Мы обсудим использование метода ignore() класса cin для устранения этой проблемы в главе 12 «Потоки и файлы».

Считывание нескольких строк

Мы смогли решить проблему считывания внутренних пробелов в строке, но как быть с несколькими строками? Оказывается, что метод cin::get() может иметь третий аргумент, который пригодится нам в этой ситуации. Этот аргумент определяет символ, на котором метод завершает считывание строки. Установленным по умолчанию значением этого аргумента является символ новой строки ('\n'), но если вы вызовете метод с другим аргументом, то это значение заменится на введенный вами символ.

В нашем следующем примере, LINESIN, мы вызовем метод cin::get() с символом доллара ('$') в качестве третьего аргумента:

```
// linesin.cpp
// ввод нескольких строк
#include <iostream>
using namespace std;
```

```
const int MAX = 2000; // максимальная длина строки
char str [ MAX ];     // сама строка
//////////////////////////////////////////////////////////
int main ( )
{
  cout << "\nВведите строку:\n";
  cin.get ( str, MAX, '$' );
  cout << "Вы ввели:\n" << str << endl;
  return 0;
}
```

Теперь вы можете напечатать столько строк для ввода, сколько хотите. Метод будет принимать символы до тех пор, пока вы не введете завершающий символ (или до тех пор, пока введенные данные не превысят размер массива). Помните, что вам нужно будет нажать клавишу Enter после того, как вы напечатаете символ '$'. Вот пример взаимодействия с программой:

```
Введите строку:
Широка страна моя родная
Много в ней лесов, полей и рек
Я другой такой страны не знаю,
Где так вольно дышит человек.
$
Вы ввели:
Широка страна моя родная
Много в ней лесов, полей и рек
Я другой такой страны не знаю,
Где так вольно дышит человек.
```

Мы заканчивали каждую строку нажатием клавиши Enter, но программа продолжала принимать от нас ввод до тех пор, пока мы не ввели символ '$'.

Копирование строк

Лучшим способом понять истинную природу строк является произведение действий с ними символ за символом. Наша следующая программа делает именно это.

```
// strcopy1.cpp
// копирование строки с использованием цикла
#include <iostream>
#include <cstring>
using namespace std;
//////////////////////////////////////////////////////////
int main ( )
{
  // исходная строка
  char str1 [ ] = "Маленькой елочке холодно зимой.";

  const int MAX = 80; // максимальная длина строки
  char str2 [ MAX ];  // сама строка

  for ( int j = 0; j < strlen ( str1 ); j++ ) // копируем strlen ( str1 )
    str2 [ j ] = str1 [ j ];                  // символов из str1 в str2
  str2 [ j ] = '\0';                          // завершаем строку нулем
```

```
  cout << str2 << endl;                    // и выводим на экран

  return 0;
}
```

Эта программа создает строковую константу str1 и строковую переменную str2. Затем в цикле for происходит копирование строковой константы в строковую переменную. За каждую итерацию цикла копируется один символ в строке программы

```
str2 [ j ] = str1 [ j ];
```

Вспомним, что компилятор объединяет две смежные строковые константы в одну, что позволяет нам записать цитату в двух строках.

Эта программа также знакомит нас с библиотечными функциями для строк. Так как в C++ нет встроенных операторов для работы со строками, то обычно используют эти функции. К счастью, их много. Одну из них мы использовали в программе strlen(), она определяет длину строки (сколько символов строка содержит). Мы используем длину строки для ограничения цикла for, чтобы скопировать нужное нам количество символов. Для использования библиотечных функций для строк нужно подключить к программе заголовочный файл CSTRING (или STRING.H).

Скопированная версия строки должна заканчиваться нулевым символом. Однако длина строки, возвращаемая функцией strlen(), не включает в себя нулевой символ. Мы могли скопировать один добавочный символ, но проще вставить нулевой символ явно. Мы делаем это в строке

```
str2 [ j ] = '\0';
```

Если вы не вставите этот символ, то затем увидите, что в строку, напечатанную программой, будут включены символы, следующие за нужной вам строкой. Операция << просто печатает все символы до тех пор, пока не встретит '\0'.

Копирование строк более простым способом

Конечно, вам не обязательно использовать цикл for для копирования строк. Как вы могли догадаться, для этого существует библиотечная функция. Приведем измененную версию предыдущей программы, STRCOPY2, которая использует функцию strcpy().

```
// strcopy2.cpp
// копирование строки функцией strcpy ( )
#include <iostream>
#include <cstring>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  char str1 [ ] = "Уронили мишку на пол, оторвали мишке лапу!";

  const int MAX = 80;   // максимальная длина строки
```

```
  char str2 [ MAX ]:    // сама строка

  strcpy( str2, str1 ); // копируем строку

  cout << str2 << endl; // и показываем результат

  return 0;
}
```

Заметим, что первым аргументом этой функции является строка, куда будут копироваться данные:

```
strcpy ( destination, source ):
```

Порядок записи справа налево напоминает формат обыкновенного присваивания: переменная справа копируется в переменную слева.

Массивы строк

Если возможно существование массива массивов, значит возможен и массив строк. На самом деле это довольно полезная конструкция. Рассмотрим пример STRARAY, в котором названия дней недели помещаются в массив:

```
// straray.cpp
// массив строк
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  const int DAYS = 7; // количество строк в массиве
  const int MAX = 12; // максимальная длина каждой из них

  // массив строк
  char star [ DAYS ][ MAX ] =
  {
    "Понедельник", "Вторник", "Среда", "Четверг",
    "Пятница", "Суббота", "Воскресенье"
  };

  // вывод всех строк на экран
  for ( int j = 0; j < DAYS; j++ )
    cout << star [ j ] << endl;

  return 0;
}
```

Программа печатает все строки массива:

```
Понедельник
Вторник
Среда
Четверг
Пятница
Суббота
Воскресенье
```

Так как строки — это массивы, то будет верным утверждение, что star — массив строк — в действительности является двумерным массивом. Его первая размерность, DAYS, определяет, сколько строк в массиве. Вторая размерность, MAX, определяет максимальную длину строк (11 символов для строки «воскресенье» плюс 12-й завершающий нулевой символ). На рисунке 7.10 показано, как это выглядит.

Заметим, что некоторое количество памяти не будет использоваться в случае, когда строка меньше максимальной длины. Мы объясним, как избежать этого недостатка, когда будем говорить об указателях.

10

7		0	1	2	3	4	5	6	7	8	9	
	0	S	u	n	d	a	y	0				star[0]
	1	M	o	n	d	a	y	0				star[1]
	2	T	u	e	s	d	a	y	0			star[2]
	3	W	e	d	n	e	s	d	a	y	0	star[3]
	4	T	h	u	r	s	d	a	y	0		star[4]
	5	F	r	i	d	a	y	0				star[5]
	6	S	a	t	u	r	d	a	y	0		star[6]

Рис. 7.10. Массив строк

Синтаксис для доступа к одной из строк массива может выглядеть немного неожиданно:

```
star [ j ]:
```

Если мы работаем с двумерным массивом, то где тогда второй индекс? Так как двумерный массив — это массив массивов, то мы можем получить доступ к элементам «внешнего» массива, каждый из которых в отдельности является массивом (в нашем случае строкой). Для того чтобы сделать это, нам не нужен второй индекс. Поэтому star[j] — это строка под номером j из массива строк.

Строки как члены классов

Строки часто используют в качестве членов классов. В следующем примере, вариации программы OBJPART из главы 6, используются строки, содержащие названия частей изделий.

```
// strpart.cpp
// использование строк в классе
#include <iostream>
#include <cstring>
using namespace std;
///////////////////////////////////////////////////////////
class part
{
```

```
  private:
    char partname [ 30 ]; // название
    int partnumber;       // номер
    double cost;          // цена
  public:
    void setpart ( char pname [ ], int pn, double c )
      {
        strcpy ( partname, pname );
        partnumber = pn;
        const = c;
      }
    void showpart ( )
      {
        cout << "\nНазвание=" << partname;
        cout << ", номер=" << partnumber;
        cout << ", цена=$" << const;
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  part part1, part2;

  part1.setpart ( "муфта", 4473, 217.55 );
  part2.setpart ( "вороток", 9924, 419.25 );
  cout << "\nПервая деталь: "; part1.showpart ( );
  cout << "\nВторая деталь: "; part2.showpart ( );
  cout << endl;

  return 0;
}
```

В этой программе определены два объекта класса part, и им присвоены значения в методе setpart(). Затем программа выводит эти объекты, используя метод showpart(). Приведем вывод:

```
Первая деталь:
Название=муфта, номер=4473, цена=$217.55
Вторая деталь:
Название=вороток, номер=9924, цена=$419.25
```

Чтобы уменьшить размер программы, мы удалили номер модели из переменных класса.

В методе setpart() мы используем библиотечную функцию strcpy() для копирования строки из аргумента pname в переменную класса partname. Таким образом, этот метод служит для строковой переменной для тех же целей, что и операция присваивания для простых переменных. (Похожая функция strncpy() принимает еще третий аргумент, определяющий максимальное количество символов, которое может быть скопировано. Так можно предупредить переполнение массива.)

Кроме изученных нами, есть еще библиотечные функции для добавления одной строки к другой, сравнения строк, поиска определенного символа в строке и выполнения многих других операций. Описание этих функций вы сможете найти в документации к вашему компилятору.

Определенные пользователем типы строк

При использовании строкового типа в C++ иногда возникают проблемы. Одна из них заключается в том, что вы не можете использовать совершенно разумное выражение

```
strDest = strSrc;
```

для установки равенства одной строки другой. (В некоторых языках, таких, как BASIC, это замечательно работает.) В стандартном классе C++ string, который мы рассмотрим в следующем разделе, эта проблема решена, но на данный момент давайте посмотрим, как мы сможем использовать технологии ООП для самостоятельного решения этой проблемы. Создадим наш собственный класс string, что даст нам возможность представлять строки как объекты класса, который прольет свет на операции стандартного класса string.

Если мы определим наш собственный тип строк, используя класс, то мы сможем использовать выражения присваивания. (Многие другие операции со строками, такие, как объединение, могут быть также упрощены, но мы должны подождать до главы 8 «Перегрузка операций», чтобы увидеть, как это сделать.)

Программа STROBJ создает класс String. (Не путайте этот самопальный класс String со стандартным встроенным классом string, название которого начинается с маленькой буквы 's'.) Приведем листинг:

```
// strobj.cpp
// строка как класс
#include <iostream>
#include <cstring>
using namespace std;
///////////////////////////////////////////////////////////
class String
{
  private:
    enum { SZ = 80; };          // максимальный размер строки
    char str [ SZ ];            // сама строка
  public:
    String ( ) // конструктор без параметров
      { str [ 0 ] = '\0'; }
    String ( char s [ ] )       // конструктор с одним параметром
      { strcpy ( str, s ); }
    void display ( )            // показ строки
      { cout << str; }
    void concat ( String s2 ) // сложение строк
      {
        if ( strlen ( str ) + strlen ( s2.str ) < SZ )
          strcat ( str, s2.str );
        else
          cout << "\nПереполнение!";
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  String s1 ( "С Новым годом!  " ); // конструктор с одним параметром
  String s2 = "С новым счастьем!";  // аналогично, но в другой форме
```

```
  String s3;                        // конструктор без параметров

  cout << "\ns1="; s1.display ( );  // показываем все строки
  cout << "\ns2="; s2.display ( );
  cout << "\ns3="; s3.display ( );

  s3 = s1;                          // присвоение
  cout << "\ns3="; s3.display ( );

  s3.concat ( s2 );                 // сложение
  cout << "\ns3="; s3.display ( );
  cout << endl;

  return 0;
}
```

Класс String содержит массив типа char. Может показаться, что наш заново определенный класс — это то же самое, что и определение строки: массив типа char. Но, поместив массив в класс, мы получили некоторую выгоду. Так как объекту может быть присвоено значение другого объекта того же класса, используя операцию =, то мы можем использовать выражения типа

```
s3 = s1;
```

как мы сделали в функции main ( ), для установки одного объекта класса String равным другому объекту. Мы можем также определить наши собственные методы для работы с объектами класса String.

В программе STROBJ все строки имеют одинаковую длину: SZ символов (SZ установлено равным 80). В ней есть два конструктора. В первом начальный символ в str устанавливается равным '\0', чтобы длина строки была равной 0. Этот конструктор вызывается в строке

```
String s3;
```

Второй конструктор присваивает объекту класса String значение строковой константы. Он использует библиотечную функцию strcpy() для копирования строковой константы в переменную объекта. Конструктор вызывается в строке

```
String s1 ( "С Новым годом!  " );
```

Альтернативный формат вызова этого конструктора, работающий с любым одноаргументным конструктором, будет таким

```
String s1 = "С Новым годом!  ";
```

Какой бы формат вы ни использовали, этот конструктор эффективно преобразует строковый тип в объект класса String, то есть обычную строковую константу в объект. Метод display() выводит строку объекта на экран.

Другой метод нашего класса String, concat(), объединяет один объект класса String с другим (складывает две строки). Первоначальный объект вызывает метод, то есть к его строке будет добавлена строка объекта, переданного в качестве аргумента. Таким образом, строка функции main()

```
s3.concat ( s2 );
```

служит для прибавления строки s2 к уже существующей строке s3. Так как s2 инициализирована как «С новым счастьем!», а s3 было присвоено значение s1, «С Новым годом!», то в результате значением s3 будет «С Новым годом! С новым счастьем!»

Метод concat() использует библиотечную функцию языка C strcat() для выполнения объединения строк. Эта библиотечная функция присоединяет строку второго аргумента к строке первого аргумента. Вывод программы будет таким:

```
s1=C Новым годом!
s2=C новым счастьем!
s3=          - сейчас строка пуста
s3=C Новым годом!           - аналог s1
s3=C Новым годом!  С новым счастьем!         - после добавления s2
```

Если длина двух строк, объединенных методом concat(), превысит максимальную длину строки для объекта String, то объединение не будет выполнено, а пользователь получит сообщение об ошибке.

Мы рассмотрели простой строковый класс. Теперь мы рассмотрим более сложную версию того же класса.

Стандартный класс string языка C++

Стандартный язык C++ включает в себя новый класс, называемый string. Этот класс во многом улучшает традиционный строковый тип. Например, вам не нужно заботиться о создании массива нужного размера для содержания строковых переменных. Класс string берет на себя всю ответственность за управление памятью. Кроме того, этот класс позволяет использовать перегруженные операции, поэтому вы можете объединять строковые объекты, используя операцию +:

```
s3 = s1 + s2;
```

Есть и другая выгода. Этот класс более эффективен и безопасен в использовании, чем строковый тип. Во многих ситуациях предпочтительнее использование класса. (Однако, как мы заметили ранее, существует и много ситуаций, в которых должен быть использован строковый тип.) В этом разделе мы рассмотрим класс string и его различные операции и методы.

Определение объектов класса string и присваивание им значений

Вы можете определить объект класса string различными способами: использовать конструктор без аргументов, создающий пустую строку, конструктор с одним аргументом, где аргумент является строковой константой, то есть символы ограничены двойными кавычками. Как и в написанном нами классе String, объекты класса string могут быть присвоены один другому при использовании простой операции присваивания. В примере SSTRASS показано, как это выглядит.

```
// sstrass.cpp
// определение и присвоение для строк
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  string s1 ( "Рыба" );
  string s2 = "Мясо";
  string s3;

  s3 = s1;
  cout << "s3 = " << s3 << endl;

  s3 = "Ни " + s1 + " ни ";
  s3 += s2;
  cout << "s3 = " << s3 << endl;

  s1.swap ( s2 );
  cout << s1 << " не " << s2 << endl;

  return 0;
}
```

Здесь в первых трех строках кода показаны три способа определения объекта класса string. Первые два инициализируют строки, последний создает пустую переменную класса string. В следующей строке показано простое присваивание с использованием операции =.

Класс string использует перегруженные операции. Мы не будем изучать внутреннюю работу перегруженных операций до следующей главы, но можем использовать эти операции и без знания того, как они сконструированы.

Перегруженная операция + объединяет один строковый объект с другим. В строке

```
s3 = "Ни " + s1 + " ни ";
```

строка «Ни» помещается в переменную s3.

Вы можете также использовать операцию += для добавления строки в конец существующей строки. В строке

```
s3 += s2;
```

переменная s2, имеющая значение «Мясо», добавляется в конец строки s3. Получившаяся при этом строка «Ни Мясо» присваивается переменной s3.

В этом примере нам встретится первый метод класса string — swap(), который меняет местами значения двух строковых объектов. Он вызывается для одного объекта, в то время как другой объект является его аргументом. Мы применили этот метод к объектам s1 («Рыба») и s2 («Мясо»), а затем вывели значения этих объектов для того, чтобы показать, что s1 теперь имеет значение «Мясо», а s2 имеет значение «Рыба». Вот результат работы программы SSTRASS:

```
s3 = Рыба
s3 = Ни Рыба ни Мясо
Мясо не Рыба
```

Ввод/вывод для объектов класса string

Ввод и вывод осуществляются путем, схожим с применяемым для строкового типа. Операции << и >> перегружены для использования с объектами класса string, метод getline() принимает ввод, который может содержать пробелы или несколько строк. В примере SSTRIO показано, как это выглядит.

```
// sstrio.cpp
// ввод/вывод для класса string
#include <iostream>
#include <string>
using namespace std:
//////////////////////////////////////////////////////////////
int main ( )
{
  string full_name. nickname. address:
  string greeting ( "Здравствуйте. " ):

  cout << "Введите Ваше имя: ":
  getline ( cin. full_name ):
  cout << "Ваше имя: " << full_name << endl:

  cout << "Введите Ваш псевдоним: ":
  cin >> nickname:

  greeting += nickname:
  cout << greeting << endl:

  cout << "Введите Ваш адрес в несколько строк\n":
  cout << "Окончание ввода символ '$'\n":
  getline ( cin. address. '$' ):
  cout << "Ваш адрес: " << address << endl:

  return 0:
}
```

Программа считывает имя пользователя, которое, возможно, содержит внутренние пробелы, используя метод getline(). Эта функция похожа на метод get(), используемый для строкового типа, но это не метод класса. Ее первым аргументом является потоковый объект, из которого будет приниматься ввод (здесь это cin), а второй аргумент — это объект класса string, куда будет помещен текст. Переменная full_name будет затем выведена, используя cout и операцию <<.

Затем программа считывает псевдоним пользователя, который предположительно состоит из одного слова, используя cin и операцию >>. Наконец, программа использует вариант функции getline() с тремя аргументами для считывания адреса пользователя, в котором может быть несколько строк. Третий аргумент функции определяет символ, который используется для завершения ввода. В программе мы использовали символ '$', который пользователь должен ввести, как последний символ строки перед тем, как нажать клавишу Enter. Если функция не имеет третьего аргумента, то предполагается, что завершающим символом строки будет '\n', представляющий клавишу Enter. Приведем пример взаимодействия с программой SSTRIO:

```
Введите Ваше имя: Джек Восьмеркин
Ваше имя: Джек Восьмеркин
Введите Ваш псевдоним: Американец
Здравствуйте, Американец
Введите Ваш адрес в несколько строк
Окончание ввода символ '$'
123456, Россия
г. Урюпинск, ул. Канавная$
Ваш адрес:
123456, Россия
г. Урюпинск, ул. Канавная$
```

Поиск объектов класса string

Класс string включает в себя различные методы для поиска строк и фрагментов строк в объектах класса string. В программе SSTRFIND показаны некоторые из них.

```
// sstrfind.cpp
// поиск подстрок
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  string s1 = "В лесу родилась елочка, в лесу она росла.";
  int n;

  n = s1.find ( "елочка" );
  cout << "Елочка найдена: " << n << endl;

  n = s1.find_first_of ( "умка" );
  cout << "Первый из умка: " << n << endl;

  n = s1.find_first_not_of ( "абвгдАБВГД" );
  cout << "Первый не из : " << n << endl;

  return 0;
}
```

Метод find() предназначен для поиска строки, используемой в качестве аргумента, в строке, для которой был вызван метод. Здесь мы ищем слово «елочка» в строке s1, которая содержит отрывок известной песенки. Оно найдено на позиции 16. Как в строковом типе, позиция самого левого символа нумеруется как 0.

Метод find_first_of() предназначен для поиска любого символа из группы и возвращает позицию первого найденного. Здесь ищем любой символ из группы 'у', 'м', 'к', 'а'. Первым из них будет найден символ 'у' в слове 'лесу' на позиции 5.

Похожий метод find_first_not_of() ищет первый символ в строке, который не входит в определенную группу символов. Здесь группа состоит из нескольких букв, прописных и строчных, поэтому функция находит первый символ пробе-

ла, который является вторым символом в строке. Вывод программы SSTRFIND будет таким:

```
Елочка найдена: 16
Первый из умка: 5
Первый не из абвгдАБВГД: 1
```

У многих из этих методов существуют варианты, которые мы не будем демонстрировать здесь, такие, как функция rfind(), переворачивающая строку; find_last_of(), ищущая последний символ, совпадающий с группой заданных символов, и find_last_not_of(). Все эти функции возвращают −1, если цель не найдена.

Модификация объектов класса string

Существуют различные пути модификации объектов класса string. В нашем следующем примере показаны методы erase(), replace() и insert() в работе.

```
// sstrchng.cpp
// изменение частей строки
#include <iostream>
#include <string>
using namespace std:
///////////////////////////////////////////////////////////
int main ( )
{
  string s1 ( "Все хорошо, прекрасная маркиза." ):
  string s2 ( "принцесса" ):
  string s3 ( "Приветствую " ):

  s1.erase ( 0. 12 ):               // убираем "Все хорошо, "
  s1.replace ( 11. 7. s2 ):         // заменяем "маркиза" на "принцесса"
  s1.insert ( 0. s3 ):              // вставляем "Приветствую, "
  s1.erase ( s1.size ( ) - 1. 1 ):  // убираем '.'
  s1.append ( 3. '!' ):             // добавляем '!!!'

  int x = s1.find ( ' ' ):          // ищем пробелы
  while ( x < s1.size ( ) )         // цикл по всем пробелам
  {
    s1.replace ( x. 1. '/' ):       // заменяем пробел на '/'
    x = s1.find ( ' ' ):            // ищем следующий пробел
  }

  cout << "s1: " << s1 << endl;

  return 0:
}
```

Метод erase() удаляет фрагмент из строки. Его первым аргументом является позиция первого символа фрагмента, а вторым — длина фрагмента. В нашем примере удалено "Все хорошо, " из начала строки. Метод replace() заменяет часть строки на другую строку. Его первым аргументом является позиция начала замены, вторым — количество символов исходной строки, которое должно быть заменено, а третьим аргументом является строка для замены. Здесь маркиза заменяется на принцесса.

Метод insert() вставляет строку, определенную во втором аргументе, на место, определенное в первом аргументе. В нашем примере "Приветствую" вставлено в начало строки s1. При втором использовании метода erase() применен метод size(), который возвращает количество символов в объекте класса string. Выражение size()-1 — это позиция последнего символа фрагмента, который будет удален. Метод append() ставит три восклицательных знака в конце предложения. В этой версии метода первый аргумент — это количество символов, которое будет добавлено, а второй аргумент — это символы, которые будут добавлены.

В конце программы мы показали идиому, которую вы можете использовать для перезаписи нескольких фрагментов в другой строке. Здесь в цикле while мы ищем символ пробела, используя метод find(), и заменяем каждый из них на слэш, используя функцию replace().

Мы начали с объекта s1, содержащего в себе строку «Все хорошо, прекрасная маркиза». После замены вывод программы SSTRCHNG будет таким:

```
s1: Приветствую/прекрасная/принцесса
```

Сравнение объектов класса string

Можно использовать перегруженные операции или метод compare() для сравнения объектов класса string. Задача состоит в том, чтобы определить, являются ли строки идентичными или какая из них предшествует другой в алфавитном порядке. В программе SSTRCOM показаны некоторые возможности.

```
// sstrcom.cpp
// сравнение строк
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  string aName = "Иван";
  string userName;

  cout << "Введите Ваше имя: ";
  cin >> userName;
  if ( userName == aName )
    cout << "Привет, Иван\n";
  else if ( userName < aName )
    cout << "Ваше имя идет до имени Иван\n";
  else
    cout << "Ваше имя идет после имени Иван\n";

  int n = userName.compare ( 0, 2, aName, 0, 2 );
  cout << "Первые две буквы Вашего имени ";
  if ( n == 0 )
    cout << "совпадают ";
  else if ( n < 0 )
    cout << "идут до ";
  else
    cout << "идут после ";
```

```
  cout << aName.substr ( 0, 2 ) << endl;

  return 0;
}
```

В первой части программы операции == и < используются для определения того, является ли написанное пользователем имя идентичным имени Иван или оно предшествует или следует за ним в алфавитном порядке. Во второй части программы метод compare() сравнивает только первые две буквы слова «Иван» с первыми двумя буквами имени, напечатанного пользователем (userName). Аргументами этой версии метода compare() являются начальная позиция userName, число символов, которые надо сравнить, строка, используемая для сравнения (aName), а также начальная позиция и количество символов в строке aName. Вот пример взаимодействия с программой SSTRCOM:

```
Введите Ваше имя: Алексей
Ваше имя идет до имени Иван
Первые две буквы Вашего имени идут до Ив
```

Первые две буквы имени «Иван» получают, используя метод substr(). Он возвращает фрагмент строки, для которой метод был вызван. Его первый аргумент — это позиция фрагмента, а второй — количество символов.

Доступ к символам в объектах класса string

Доступ к отдельным символам объектов класса string вы можете получить разными способами. В нашем следующем примере мы покажем доступ с использованием метода at(). Вы можете также использовать перегруженную операцию [ ], которая позволяет рассматривать объект класса string как массив. Однако операция [ ] не предупредит вас, если вы попытаетесь получить доступ к символу, лежащему за пределами массива (например, после конца строки). Операция [ ] ведет себя здесь так, как обращается с настоящим массивом, и это более эффективно. Но это может привести к тяжелым для диагностики программным ошибкам. Безопасней использовать метод at(), который остановит программу, если вы используете индекс, не входящий в допустимые границы. (Есть одно исключение, и мы обсудим его в главе 14 «Шаблоны и исключения».)

```
// sstrchar.cpp
// доступ к символам в строке
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
int main ( )
{
  char charray [ 80 ];
  string word;

  cout << "Введите слово: ";
  cin >> word;
  int wlen = word.length ( );      // длина строки
```

```
  cout << "По одному символу: ";
  for ( int j = 0; j < wlen; j++ )
    cout << word.at ( j );        // тут будет проверка на выход за пределы строки
    // cout << word [ j ];        // а тут проверки не будет

  word.copy ( charray, wlen, 0 ); // копируем строку в массив
  charray [ wlen ] = 0;
  cout << "\nМассив содержит: " << charray << endl;

  return 0;
}
```

В этой программе мы использовали метод at() для вывода содержимого объекта класса string символ за символом. Аргумент метода at() — это местонахождение символа в строке.

Затем мы показали, как вы можете использовать метод copy() для копирования объекта класса string в массив типа char, эффективно преобразовывая его в строковый тип. Вслед за копированием, после последнего символа строки должен быть вставлен нулевой символ ('\0') для завершения преобразования к строковому типу. Метод length() класса string возвращает то же число, что и метод size(). Приведем вывод программы SSTRCHAR:

```
Введите слово: симбиоз
По одному символу: симбиоз
Массив содержит: симбиоз
```

(Вы можете также преобразовать объект класса string к строковому типу, используя методы c_str() или data(). Однако для использования этих методов вам нужно изучить указатели, которые мы рассмотрим в главе 10.)

Другие методы класса string

Мы видели, что методы size() и length() возвращают число символов строки объекта класса string. Количество памяти, занятое строкой, обычно больше, чем в действительности это необходимо. (Хотя если строка не была инициализирована, то она использует для символов 0 байтов.) Метод capacity() возвращает действительное количество занятой памяти. Вы можете добавлять символы в строку, не добавляя памяти в нее до тех пор, пока этот лимит не будет исчерпан. Метод max_size() возвращает максимально возможный размер объекта класса string. Это количество будет на три байта меньше, чем максимальное значение переменной типа int в вашей системе. В 32-битной Windows системе это 4 294 967 293 байта, но размер памяти будет, возможно, ограничен этим количеством.

Многие из методов класса string, которые мы обсудили, имеют варианты, отличающиеся количеством и типом аргументов. Обратитесь к документации вашего компилятора для выяснения деталей.

Вы должны знать, что объекты класса string не заканчиваются нулевым символом, как это происходит в строковом типе. Вместо этого у нас есть переменная класса, хранящая длину строки. Поэтому, если вы будете просматривать строку, то не надейтесь найти в ней нулевой символ, обозначающий конец строки.

Класс string — в действительности только один из возможных строковых классов, производных от класса-шаблона basic_string. Класс string основан на типе char, но общий вариант использует тип wchar_t. Это позволяет классу basic_string использоваться при работе с другими языками, имеющими больше символов, чем в русском. Файлы помощи вашего компилятора могут содержать список функций класса string в классе basic_string.

Резюме

Массивы содержат набор данных одинакового типа. Этот тип может быть простым типом, структурой или классом. Члены массива называются элементами. К элементам можно получить доступ, используя число, которое называется индексом. Элементы могут быть инициализированы определенным значением при определении массива. Массив может иметь несколько размерностей. Двумерный массив — это массив массивов. Адрес массива может быть использован как аргумент функции; сам массив при этом не копируется. Массив может быть использован как переменная класса. Необходимо позаботиться о том, чтобы данные не были помещены за пределы массива.

Строковый тип представляет собой массив элементов типа char. Последний символ такой строки должен быть нулевым, '\0'. Строковая константа имеет специальную форму, она может быть записана удобным для нас способом: это текст, заключенный в двойные кавычки. Для работы со строками используются различные библиотечные функции. Массив строк — это массив массивов типа char. Создавая строковую переменную, нужно быть уверенным, что массив имеет достаточный размер для помещения в него строки. Строки, используемые в качестве аргументов в библиотечных функциях, вы сможете найти в старых программах. Обычно такое использование строкового типа в современных программах не рекомендуется.

Предпочтительнее использовать для работы со строками объекты класса string. Эти объекты могут работать с перегруженными операциями и методами класса. Пользователю не нужно заботиться об управлении памятью при использовании объектов класса string.

Вопросы

Ответы на эти вопросы вы сможете найти в приложении Ж.

1. Доступ к элементам массива осуществляется с помощью:
   а) подхода FIFO;
   б) операции точки;
   в) имени элемента;
   г) индекса элемента.

2. Все элементы массива должны быть ________ типа.
3. Напишите выражение, которое определяет одномерный массив, именованный как doubleArray, типа double, содержащий 100 элементов.
4. Элементы 10-элементного массива нумеруются начиная с _______ и до __ _____.
5. Напишите выражение, которое выводит j элемент массива doubleArray с помощью cout и операции <<.
6. Какой по счету элемент массива doubleArray[7]?
   а) шестой;
   б) седьмой;
   в) восьмой;
   г) неизвестно.
7. Напишите выражение, которое определяет массив coins типа int и инициализирует его значениями пенни: 5 центов, 10 центов, 25 центов, 50 центов и 1 доллар.
8. При доступе к многомерному массиву его индексы:
   а) разделены запятыми;
   б) заключены в квадратные скобки и разделены запятыми;
   в) разделены запятыми и заключены в квадратные скобки;
   г) заключены в квадратные скобки.
9. Напишите выражение для доступа к 4-му элементу 2-го подмассива двумерного массива twoD.
10. Истинно ли следующее утверждение: в C++ возможна реализация четырехмерного массива?
11. Для двумерного массива flarr типа float запишите выражение, которое объявляет массив и инициализирует его первый подмассив значениями 52, 27, 83; второй — значениями 94, 73, 49; третий — значениями 3, 6, 1.
12. Имя массива, используемое в файлах кода, представляет собой ________ массива.
13. При передаче имени массива в функцию она:
    а) работает с тем же массивом, с которым работает и вызывающая функцию программа;
    б) работает с копией массива, переданной программой;
    в) ссылается на массив, используя то же имя, которое используется в вызывающей программе;
    г) ссылается на массив, используя другое имя, чем то, которое используется в вызывающей программе.

14. Что определяет это выражение?

```
employee emplist [ 1000 ];
```

15. Напишите выражение для доступа к переменной salary структуры, которая является 17-м элементом массива emplist.
16. Данные, помещенные в стек первыми:
    а) не имеют индексного номера;
    б) имеют индекс, равный 0;
    в) будут первыми извлечены из стека;
    г) будут извлечены из стека последними.
17. Напишите выражение, которое определяет массив manybirds, содержащий в себе 50 объектов типа bird.
18. Истинно ли следующее утверждение: компилятор будет протестовать, если вы попытаетесь получить доступ к 14 элементу массива в 10-элементном массиве?
19. Напишите выражение, которое вызывает метод cheep() для объекта класса bird, являющегося 27-м элементом массива manybirds.
20. Строка в C++ — это ____________ типа ____________.
21. Напишите выражение, которое определяет строковую переменную city, содержащую строку длиной до 20 символов (это небольшая хитрость).
22. Напишите выражение, которое определяет строковую константу dextrose, имеющую значение «C6H12O6 – H2O».
23. Истинно ли следующее утверждение: операция >> прекращает считывание строки при обнаружении пробела?
24. Вы можете считывать ввод, который содержит несколько строк или текст, используя:
    а) обыкновенную комбинацию cout <<;
    б) метод cin.get() с одним аргументом;
    в) метод cin.get() с двумя аргументами;
    г) метод cin.get() с тремя аргументами;
    д) метод cin.get() с одним аргументом.
25. Напишите выражение, которое использует библиотечную функцию для копирования строки name в строку blank.
26. Напишите объявление класса dog, который содержит две переменных: строку breed и переменную age типа int (методов класс не имеет).
27. Истинно ли следующее утверждение: предпочтительнее использовать строковый тип вместо стандарного класса string в своих программах?
28. Объекты класса string:
    а) заканчиваются нулевым символом;
    б) могут быть скопированы с операцией присваивания;

в) не требуют управления памятью;

г) не имеют методов класса.

29. Напишите выражение, которое осуществляет поиск строки «кот» в строке s1.

30. Напишите выражение, которое вставляет строку «кот» в строку s1 на позицию 12.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Напишите функцию reversit(), которая переворачивает строку (массив типа char). Используйте цикл for, который меняет местами первый и последний символы, затем следующие и т. д. до предпоследнего. Строка должна передаваться в функцию reversit() как аргумент.

Напишите программу для выполнения функции reversit(). Программа должна принимать строку от пользователя, вызывать функцию reversit(), а затем выводить полученный результат. Используйте метод ввода, который позволяет использовать внутренние пробелы. Протестируйте программу на примере фразы «Аргентина манит негра».

*2. Создайте класс employee, который содержит имя (объект класса string) и номер (типа long) служащего. Включите в него метод getdata(), предназначенный для получения данных от пользователя и помещения их в объект, и метод putdata(), для вывода данных. Предполагаем, что имя может иметь внутренние пробелы.

Напишите функцию main(), использующую этот класс. Вам нужно будет создать массив типа employee, а затем предложить пользователю ввести данные до 100 служащих. Наконец, вам нужно будет вывести данные всех служащих.

*3. Напишите программу, вычисляющую среднее значение 100 интервалов, введенных пользователем. Создайте массив объектов класса Distance, как это было сделано в примере ENGLARAY этой главы. Для вычисления среднего значения вы можете позаимствовать метод add_dist() из примера ENGLCON главы 6. Вам также понадобится метод, который выделяет целую часть из значения Distance. Вот одна из возможностей:

```
void Distance::div_dist ( Distance d2, int divisor )
{
  float fltfeet = d2.feet + d2.inches / 12.0;
  fltfeet /= divisor;
  fltfeet = int ( fltfeet );
  inches = ( fltfeet - feet ) * 12.0;
}
```

4. Начните с программы, которая позволяет пользователю вводить целые числа, а затем сохранять их в массиве типа int. Напишите функцию maxint(), которая, обрабатывая элементы массива один за другим, находит наибольший. Функция должна принимать в качестве аргумента адрес массива и количество элементов в нем, а возвращать индекс наибольшего элемента. Программа должна вызвать эту функцию, а затем вывести наибольший элемент и его индекс. (Смотрите программу SALES этой главы.)

5. Начните с класса fraction из упражнений 11 и 12 главы 6. Напишите функцию main(), которая получает случайные дробные числа от пользователя, сохраняет их в массиве типа fraction, вычисляет среднее значение и выводит результат.

6. В игре бридж каждому из игроков раздают 13 карт, таким образом колода расходуется полностью. Модифицируйте программу CARDARAY этой главы так, чтобы после перемешивания колоды она делилась на четыре части по 13 карт каждая. Каждая из четырех групп карт затем должна быть выведена.

7. Одним из недостатков C++ является отсутствие для бизнес-программ встроенного типа для денежных значений, такого, как $173 698 001,32. Такой денежный тип должен иметь возможность для хранения числа с фиксированной десятичной точкой точностью около 17 знаков, которого было бы достаточно для хранения национального долга в долларах и центах. К счастью, встроенный тип C++ long double имеет точность 19 цифр, поэтому мы можем использовать его как базисный для класса money, даже используя плавающую точку. Однако нам нужно будет добавить возможность ввода и вывода денежных значений с предшествующим им знаком доллара и разделенными запятыми группы по три числа: так проще читать большие числа. Первым делом при разработке такого класса напишем метод mstold(), который принимает денежную строку, то есть строку, представляющую собой некоторое количество денег типа

   ```
   "$1,234,567,890,123.99"
   ```

   в качестве аргумента и возвращает эквивалентное ее значению число типа long double.

   Вам нужно будет обработать денежную строку как массив символов и, просматривая ее символ за символом, скопировать из нее только цифры (0–9) и десятичную точку в другую строку. Игнорируется все остальное, включая знак доллара и запятые. Затем вы можете использовать библиотечную функцию _atold() (заметим, что здесь название функции начинается с символа подчеркивания — заголовочные файлы STDLIB.H или MATH.H) для преобразования новой строки к числу типа long double. Предполагаем, что денежное значение не может быть отрицательным. Напишите функцию main() для проверки метода mstold(), которая несколько раз получает денежную строку от пользователя и выводит соответствующее число типа long double.

8. Другим недостатком C++ является отсутствие автоматической проверки индексов массива на соответствие их границам массива (это делает действия с массивами быстрыми, но менее надежными). Мы можем использовать класс для создания надежного массива, который проверяет индексы при любой попытке доступа к массиву.

   Напишите класс safearay, который использует массив типа int фиксированного размера (назовем его LIMIT) в качестве своей единственной переменной. В классе будет два метода. Первый, putel(), принимает индекс и значение типа int как аргументы и вставляет это значение в массив по заданному индексу. Второй, getel(), принимает индекс как аргумент и возвращает значение типа int, содержащееся в элементе с этим индексом.

```
safearay sa1;              // описываем массив
int temp = 12345;          // описываем целое
sa1.putel ( 7, temp );     // помещаем значение temp в массив
temp = sa1.getel ( 7 );    // получаем значение из массива
```

   Оба метода должны проверять индекс аргумента, чтобы быть уверенными, что он не меньше 0 и не больше, чем LIMIT–1. Вы можете использовать этот массив без опаски, что запись будет произведена в другие части памяти.

   Использование методов для доступа к элементам массива не выглядит так наглядно, как использование операции [ ]. В главе 8 мы увидим, как перегрузить эту операцию, чтобы сделать работу нашего класса safearay похожей на работу встроенных массивов.

9. Очередь — это устройство для хранения данных, похожее на стек. Отличие в том, что в стеке последний сохраненный элемент будет первым извлеченным, тогда как в очереди первый сохраненный элемент будет первым извлеченным. То есть в стеке используется подход «последний вошел — первый вышел» (LIFO), а в очереди используется подход «первый вошел — первый вышел» (FIFO). Очередь похожа на простую очередь посетителей магазина: первый, кто встал в очередь, будет обслужен первым.

   Перепишите программу STAKARAY из этой главы, включив в нее класс queue вместо класса stack. Кроме того, класс должен иметь два метода: один, называемый put(), для помещения элемента в очередь; и другой, называемый get(), для извлечения элемента из очереди. Эти методы эквивалентны методам push() и pop() класса stack.

   Оба класса, stack и queue, используют массив для хранения данных. Однако вместо одного поля top типа int, как в классе stack, вам понадобятся два поля для очереди: одна, называемая head, указывающая на начало очереди; и вторая, tail, указывающая на конец очереди. Элементы помещаются в конец очереди (как посетители банка, становящиеся в очередь), а извлекаются из начала очереди. Конец очереди перемещается к началу по массиву по мере того, как элементы добавляются и извлекаются из очереди. Такие результаты добавляют сложности: если одна из двух переменных

head или tail примут значение конца массива, то следует вернуться на начало. Таким образом, вам нужно выражение типа

```
if ( tail ==MAX - 1 )
  tail = -1;
```

для возврата переменной tail и похожее выражение для возврата переменной head. Массив, используемый в очереди, иногда называют круговым буфером, так как начало и конец очереди циркулируют по нему вместе с ее данными.

10. Матрица — это двумерный массив. Создайте класс matrix, который предоставляет те же меры безопасности, как и класс из упражнения 7, то есть осуществляет проверку индексов массива на вхождение их в границы массива. Полем класса matrix будет массив 10 на 10. Конструктор должен позволять программисту определить реальный размер массива (допустим, сделать его меньше, чем 10 на 10). Методам, предназначенным для доступа к членам матрицы, теперь нужны два индекса: по одному для каждой размерности массива. Вот фрагмент функции main(), которая работает с таким классом:

```
matrix m1 ( 3, 4 );          // описываем матрицу
int temp = 12345;            // описываем целое
m1.putel ( 7, 4, temp );  // помещаем значение temp в матрицу
temp = m1.getel ( 7, 4 ); // получаем значение из матрицы
```

11. Вернемся к обсуждению денежных строк из упражнения 6. Напишите метод ldtoms() для преобразования числа типа long double в денежную строку, представляющую это число. Для начала вам нужно проверить, что значение long double не очень большое. Мы предполагаем, что вы не будете пытаться преобразовать число, больше чем 9 999 999 999 999 990.00. Затем преобразуем long double в строку (без знака доллара и запятых), хранящуюся в памяти, используя объект ostrstream, как рассматривалось ранее в этой главе. Получившаяся отформатированная строка может быть помещена в буфер, называющийся ustring.

    Затем вам нужно будет создать другую строку, начинающуюся со знака доллара, далее копируем цифру за цифрой из строки ustring, начиная слева и вставляя запятые через каждые три цифры. Также вам нужно подавлять нули в начале строки. Например, вы должны вывести $3 124.95, а не $0 000 000 000 003 124.95. Не забудьте закончить строку нулевым символом '\0'.

    Напишите функцию main() для тестирования этой функции путем многократного ввода пользователем чисел типа long double и вывода результата в виде денежной строки.

12. Создайте класс bMoney. Он должен хранить денежные значения как long double. Используйте метод mstold() для преобразования денежной строки, введенной пользователем, в long double, и метод ldtoms() для преобразования числа типа long double в денежную строку для вывода (см. упражнения 6 и 10). Вы можете вызывать для ввода и вывода методы getmoney()

и putmoney(). Напишите другой метод класса для сложения двух объектов типа bMoney и назовите его madd(). Сложение этих объектов легко произвести: просто сложите переменную типа long double одного объекта с такой же переменной другого объекта. Напишите функцию main(), которая просит пользователя несколько раз ввести денежную строку, а затем выводит сумму значений этих строк. Вот как может выглядеть определение класса:

```
class bMoney
{
  private:
    long double money;
  public:
    bMoney ( );
    bMoney ( char s [ ] );
    void madd ( bMoney m1, bMoney m2 );
    void getmoney ( );
    void putmoney ( );
};
```

Глава 8
Перегрузка операций

- ✦ Перегрузка унарных операций
- ✦ Перегрузка бинарных операций
- ✦ Преобразование типов
- ✦ UML диаграммы классов
- ✦ «Подводные камни» перегрузки операций и преобразования типов
- ✦ Ключевые слова explicit и mutable

Перегрузка операций — это одна из самых захватывающих возможностей ООП. Она может превратить сложный и малопонятный листинг программы в интуитивно понятный. Например, строки

```
d3.addobjects ( d1, d2 );
```

похожие, но одинаково непонятные

```
d3 = d1.addobjects ( d2 );
```

можно заменить на более читаемую

```
d3 = d1 + d2;
```

Довольно непривлекательный термин «перегрузка операций» дается обычным операциям C++, таким, как +, *, <= или +=, в случае их применения с определенными пользователем типами данных.

Обычно

```
a = b + c;
```

работают только с основными типами данных, такими, как int или float, и попытка использования обычных операторов, когда a, b и c являются объектами определенного пользователем класса, приведет к протестам компилятора. Однако, используя перегрузку, вы можете сделать эту строку правильной даже в том случае, если a, b и c являются объектами определенного пользователем класса.

На самом деле перегрузка операций дает вам возможность переопределить язык C++. Если вы считаете, что ограничены стандартными возможностями операций, то вы можете назначить их выполнять нужные вам действия. Исполь-

зуя классы для создания новых видов переменных и перегрузку для определения новых операций, вы можете, расширив C++, создать новый, свой собственный язык.

Другой вид действий, преобразование типов, тесно связан с перегрузкой операций. C++ совершает преобразование простых типов, таких, как int или float, автоматически; но преобразование, включающее в себя созданные пользователем типы, требует немного работы со стороны программиста. Мы рассмотрим преобразование типов во второй части этой главы.

Перегрузка операций — непростая тема. Мы обсудим некоторые нюансы ее использования в конце главы.

Перегрузка унарных операций

Давайте начнем с перегрузки унарных операций. Как нам известно из главы 2, унарные операции имеют только один операнд (операнд — это переменная, на которую действует операция). Примером унарной операции могут служить операции уменьшения и увеличения ++ и -- и унарный минус, например -33.

В примере COUNTER главы 6 «Объекты и классы» мы создали класс Counter для хранения значения счетчика. Объект этого класса мы увеличивали вызовом метода:

```
c1.inc_count ( );
```

Он выполнял свою работу, но листинг программы станет более понятным, если мы применим вместо этой записи операцию увеличения ++.

```
++c1;
```

Все опытные программисты C++ (и C) сразу догадаются, что это выражение увеличивает переменную c1.

Давайте перепишем программу COUNTER, чтобы сделать это действие возможным. Приведем листинг программы COUNTPP1[1].

```
// countpp1.cpp
// увеличение переменной операцией ++
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Counter
{
  private:
    unsigned int count;        // значение счетчика
  public:
    Counter ( ) : count ( 0 )  // конструктор
      { }
    unsigned int get_count ( ) // получить значение
      { return count; }
    void operator++ ( )       // увеличить значение
      { ++count; }
};
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
///////////////////////////////////////////////////////////
int main ( )
{
  Counter c1, c2;                    // определяем переменные

  cout << "\nc1 = " << c1.get_count ( ); // выводим на экран
  cout << "\nc2 = " << c2.get_count ( );

  ++c1;                              // увеличиваем c1
  ++c2;                              // увеличиваем c2
  ++c2;                              // увеличиваем c2

  cout << "\nc1 = " << c1.get_count ( ); // снова показываем значения
  cout << "\nc2 = " << c2.get_count ( ) << endl;

  return 0;
}
```

В этой программе мы создали два объекта класса Counter: c1 и c2. Значения счетчиков выведены на дисплей, сначала они равны 0. Затем, используя перегрузку операции ++, мы увеличиваем c1 на единицу, а c2 на два и выводим полученные значения. Результат работы программы:

```
c1 = 0            – изначально обе переменные равны нулю
c2 = 0
c1 = 1            – после однократного увеличения
c2 = 2            – после двукратного увеличения
```

Выражения, соответствующие этим действиям:

```
++c1;
++c2;
++c2;
```

Операция ++ применена к c1 и дважды применена к c2. В этом примере мы использовали префиксную версию операции ++, постфиксную мы объясним позднее.

Ключевое слово operator

Как использовать обычные операции с определенными пользователями типами? В этом объявлении использовано ключевое слово operator для перегрузки операции ++:

```
void operator++ ( )
```

Сначала пишут возвращаемый тип (в нашем случае void), затем ключевое слово operator, затем саму операцию (++) и наконец список аргументов, заключенный в скобки (здесь он пуст). Такой синтаксис говорит компилятору о том, что если операнд принадлежит классу Counter, то нужно вызывать функцию с таким именем, встретив в тексте программы операцию ++.

Мы видели в главе 5 «Функции», что только компилятор может различать перегружаемые функции по типу данных и количеству их аргументов. Перегру-

жаемые операции компилятор отличает по типу данных их операндов. Если операнд имеет базовый тип, такой, как int в

```
++intvar;
```

то компилятор будет использовать свою встроенную процедуру для увеличения переменной типа int. Но если операнд является объектом класса Counter, то компилятор будет использовать написанную программистом функцию operator++().

Аргументы операции

В функции main() операция ++ применена к объекту, как в выражении ++c1. До сих пор функция operator++() не имела аргументов. Что же увеличивает эта операция? Она увеличивает переменную count объекта, к которому применяется. Так как методы класса всегда имеют доступ к объектам класса, для которых они были вызваны, то для этой операции не требуются аргументы. Это показано на рис. 8.1.

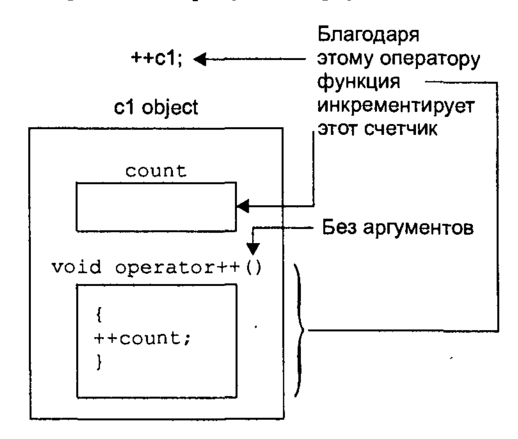


Рис. 8.1. Перегрузка унарной операции без аргументов

Значения, возвращаемые операцией

Функция operator++() программы COUNTPP1 имеет небольшой дефект. Вы можете его выявить, если используете в функции main() строку, похожую на эту:

```
c1 = ++c2;
```

Компилятор будет протестовать. Почему? Просто потому, что мы определили тип void для возвращаемого значения функции operator++(). А в нашем выражении присваивания будет запрошена переменная типа Counter. То есть компилятор запросит значение переменной c2, после того как она будет обработана операцией ++, и присвоит ее значение переменной c1. Но, при данном нами определении в COUNTPP1, мы не можем использовать ++ для увеличения объекта Counter в выражении присваивания: с таким операндом может быть использована только операция ++. Конечно, обыкновенная операция ++, примененная к данным таких типов, как int, не столкнется с этими проблемами.

Для того чтобы иметь возможность использовать написанный нами operator++() в выражениях присваивания, нам необходимо правильно определить тип его возвращаемого значения. Это сделано в следующей нашей программе COUNTPP2.

```
// countpp2.cpp
// операция ++, возвращающий значение
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Counter
{
  private:
    unsigned int count;
  public:
    Counter ( ) : count ( 0 )
      { }
    unsigned int get_count ( )
      { return count; }
    Counter operator++ ( )
      {
        ++count;
        Counter temp;
        temp.count = count;
        return temp;
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Counter c1, c2;                        // определяем переменные

  cout << "\nc1 = " << c1.get_count ( ); // выводим на экран
  cout << "\nc2 = " << c2.get_count ( );

  ++c1;                                  // увеличиваем c1
  c2 = ++c1;                             // c1=2, c2=2

  cout << "\nc1 = " << c1.get_count ( ); // снова показываем значения
  cout << "\nc2 = " << c2.get_count ( ) << endl;

  return 0;
}
```

Здесь функция operator++() создает новый объект класса Counter, названный temp, для использования его в качестве возвращаемого значения. Она сначала увеличивает переменную count в своем объекте, а затем создает объект temp и присваивает ему значение count, то же значение, что и в собственном объекте. В конце функция возвращает объект temp. Получаем ожидаемый эффект. Выражение типа

```
++c1
```

теперь возвращает значение, которое можно использовать в других выражениях, таких, как:

```
c2 = ++c1;
```

как показано в функции main(), где значение, возвращаемое c1++, присваивается c2. Результат работы этой программы будет следующим:

```
c1 = 0
c2 = 0
c1 = 2
c2 = 2
```

Временные безымянные объекты

В примере COUNTPP2 мы создали временный объект temp типа Counter, чьей единственной целью было хранение возвращаемого значения для операции ++. Реализация этого объекта потребовала трех строк программы:

```
Counter temp;       // временный объект Counter
temp.count = count; // присваиваем ему значение count
return temp;        // возвращаем объект
```

Существует много путей возвращения временного объекта из функции или перегруженной операции. Давайте рассмотрим еще один из способов в программе COUNTPP3:

```
// countpp3.cpp
// операция ++ с использованием недекларированной переменной
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Counter
{
  private:
    unsigned int count;
  public:
    Counter ( ) : count ( 0 )
      { }
    Counter ( int c ) : count ( c )
      { }
    unsigned int get_count ( )
      { return count; }
    Counter operator++ ( )
      {
        ++count;
        return Counter ( count );
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Counter c1, c2;                        // определяем переменные

  cout << "\nc1 = " << c1.get_count ( ); // выводим на экран
  cout << "\nc2 = " << c2.get_count ( );

  ++c1;                                  // увеличиваем c1
  c2 = ++c1;                             // c1=2, c2=2
```

```
  cout << "\nc1 = " << c1.get_count ( ); // снова показываем значения
  cout << "\nc2 = " << c2.get_count ( ) << endl;

  return 0;
}
```

В строке этой программы

```
return Counter ( count );
```

происходят все те же действия, которые в программе COUNTPP2 занимали три строки. Здесь создается объект типа Counter. Он не имеет имени, так как оно нигде не будет использоваться. Этот объект инициализируется значением, полученным в виде параметра count.

Но постойте: требуется ли здесь конструктор с одним аргументом? Да, требуется. Поэтому мы вставили этот конструктор в список методов класса в программе COUNTPP3.

```
Counter ( int c ) : count ( c )
  { }
```

Объект, инициализированный значением count, может быть возвращен функцией. Результат работы этой программы тот же, что и программы COUNTPP2.

В обеих программах использовано создание копии исходного объекта (объекта, для которого вызывается функция), эта копия затем и возвращается функцией. (В другом подходе применяется возврат исходного объекта с использованием указателя this; мы увидим это в главе 11 «Виртуальные функции».)

Постфиксные операции

До сих пор мы применяли операцию увеличения, используя только префиксную запись:

```
++c1
```

А как можно использовать постфиксную запись, где переменная увеличивается после того, как ее значение было использовано в выражении?

```
c1++
```

Чтобы иметь возможность работать с двумя версиями операции, мы определим два варианта перегрузки операции ++. Это показано в программе POSTFIX.

```
// postfix.cpp
// префиксная и постфиксная операции ++ для нашего класса
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Counter
{
private:
    unsigned int count;
  public:
    Counter ( ) : count ( 0 )
      { }
    Counter ( int c ) : count ( c )
```

```
    { }
  unsigned int get_count ( )
    { return count; }
  Counter operator++ ( )
    {
      return Counter ( ++count );
    }
   Counter operator++ ( int )
    {
      return Counter ( count++ );
    }
};
////////////////////////////////////////////////////////////
int main ( )
{
  Counter c1, c2;                       // определяем переменные

  cout << "\nc1 = " << c1.get_count ( ); // выводим на экран
  cout << "\nc2 = " << c2.get_count ( );

  ++c1;                                 // увеличиваем c1
  c2 = ++c1;                            // c1=2, c2=2

  cout << "\nc1 = " << c1.get_count ( ); // снова показываем значения
  cout << "\nc2 = " << c2.get_count ( );

  c2 = c1++;

  cout << "\nc1 = " << c1.get_count ( ); // и снова
  cout << "\nc2 = " << c2.get_count ( ) << endl;

  return 0;
}
```

Теперь у нас есть два типа объявления функции operator++. С одной из них, для префиксной операции, мы уже были знакомы ранее:

```
Counter operator++ ( );
```

Для реализации постфиксной записи операции ++ используем новую функцию:

```
Counter operator++ ( int );
```

Различие между этими функциями только в том, что в скобках стоит int. Здесь int не играет роли аргумента и не означает целое число. Это просто сигнал для компилятора, чтобы использовалась постфиксная версия операции. Разработчики C++ нашли полезным повторное использование существующих операций и ключевых слов; в данном случае int предназначена также и для обозначения постфиксной операции. (А сможем ли мы использовать лучший синтаксис?) Результат работы программы будет таким:

```
c1 = 0
c2 = 0
c1 = 2
c2 = 2
c1 = 3
c2 = 2
```

Первые четыре строки мы уже видели в программах COUNTPP2 и COUNTPP3. А в последних двух строках мы видим результаты, полученные после обработки выражения

```
c2 = c1++;
```

Здесь c1 увеличивается до 3, но переменной c2 значение переменной c1 было присвоено до ее увеличения. Поэтому переменная c2 имеет значение 2.

Конечно же, мы можем использовать различные варианты записи операции и для operator--.

Перегрузка бинарных операций

Бинарные операции могут быть перегружены так же, как и унарные операции. Мы рассмотрим примеры перегрузки арифметических операций, операций сравнения и операции присваивания.

Арифметические операции

В программе ENGLCON из главы 6 мы рассматривали два объекта класса Distance, которые складывались с помощью метода add_dist():

```
dist3.add_dist ( dist1, dist2 );
```

Используя перегрузку операции +, мы можем отказаться от этого неприглядного выражения и воспользоваться таким:

```
dist 3 = dist1 + dist2;
```

В листинге программы ENGLPLUS реализована эта возможность:

```
// engplus.cpp
// перегрузка операции + для сложения переменных типа Distances
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance  // класс английских мер длины
{
  private:
    int feet;
    float inches;
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ), inches ( 0.0 )
      { }
    // конструктор с двумя параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in )
      { }
    // получение информации от пользователя
    void getdist ( )
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: ";  cin >> inches;
```

```
    }
  // показ информации
  void showdist ( )
    { cout << feet << "\'-" << inches << '\"'; }
  // сложение двух длин
  Distance operator+ ( Distance ) const;
};
//////////////////////////////////////////////////////////
// сложение двух длин
Distance Distance::operator+ ( Distance d2 ) const
{
  int f = feet + d2.feet;       // складываем футы
  float i = inches + d2.inches; // складываем дюймы
  if ( i >= 12.0 )              // если дюймов стало больше 12
  {
    i -= 12.0;                  // то уменьшаем дюймы на 12
    f++;                        // и увеличиваем футы на 1
  }
  return Distance ( f, i );     // создаем и возвращаем временную переменную
}
//////////////////////////////////////////////////////////
int main ( )
{
  Distance dist1, dist3, dist4;  // определяем переменные
  dist1.getdist ( );             // получаем информацию

  Distance dist2 ( 11, 6.25 );   // определяем переменную с конкретным значением

  dist3 = dist1 + dist2;         // складываем две переменные

  dist4 = dist1 + dist2 + dist3; // складываем несколько переменных

  // показываем, что же у нас получилось
  cout << "dist1 = "; dist1.showdist ( ); cout << endl;
  cout << "dist2 = "; dist2.showdist ( ); cout << endl;
  cout << "dist3 = "; dist3.showdist ( ); cout << endl;
  cout << "dist4 = "; dist4.showdist ( ); cout << endl;

  return 0;
}
```

Для того чтобы показать, что результат сложения может быть использован в других операциях, мы выполнили в функции main() следующие действия: сложили dist1, dist2 и dist3, чтобы получить dist4 (значение которой представляет собой удвоенное значение переменной dist3), в строке:

```
dist4 = dist1 + dist2 + dist3;
```

Вот результат работы программы:

```
Введите футы: 10
Введите дюймы: 6.5

dist1 = 10'-6.5"          - ввод пользователя
dist2 = 11'-6.25"         - предопределено в программе
dist3 = 22'-0.75"         - dist1+dist2
dist4 = 44'-1.5"          - dist1+dist2+dist3
```

Объявление метода operator+() в классе Distance выглядит следующим образом:

```
Distance operator+ ( Distance );
```

Эта операция возвращает значение типа Distance и принимает один аргумент типа Distance.

В выражении

```
dist3 = dist1 + dist2;
```

важно понимать, к каким объектам будут относиться аргументы и возвращаемые значения. Когда компилятор встречает это выражение, то он просматривает типы аргументов. Обнаружив только аргументы типа Distance, он выполняет операции выражения, используя метод класса Distance operator+(). Но какой из объектов используется в качестве аргумента этой операции — dist1 или dist2? И не нужно ли нам использовать два аргумента, так как мы складываем два объекта?

Существует правило: объект, стоящий с левой стороны операции (в нашем случае dist1), вызывает функцию оператора. Объект, стоящий справа от знака операции, должен быть передан в функцию в качестве аргумента. Операция возвращает значение, которое мы затем используем для своих нужд. В нашем случае мы присваиваем его объекту dist3. Рисунок 8.2 иллюстрирует все эти действия.

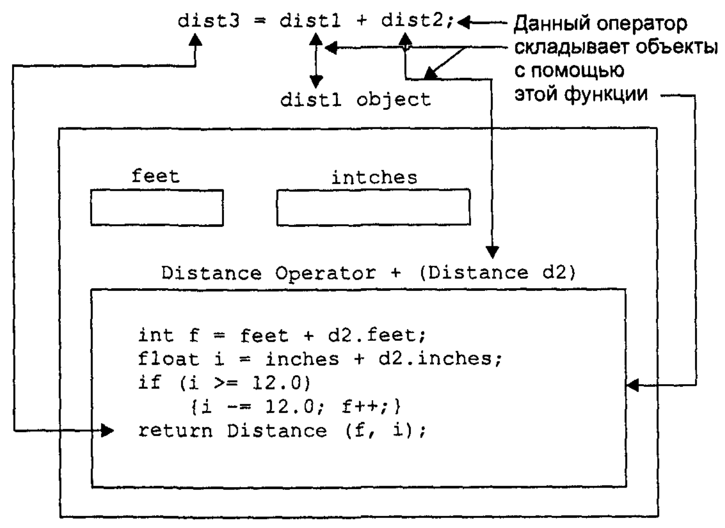


Рис. 8.2. Перегрузка бинарной операции с одним аргументом

В функции operator+() к левому операнду мы имеем прямой доступ, используя feet и inches, так как это объект, вызывающий функцию. К правому операнду мы имеем доступ как к аргументу функции, то есть как d2.feet и d2.inches.

Мы можем обобщить изложенное выше и сказать, что перегруженной операции всегда требуется количество аргументов на один меньшее, чем количество операндов, так как один из операндов является объектом, вызывающим функцию. Поэтому для унарных операций не нужны аргументы (это правило не-

верно для функции и операторов, являющихся дружественными для класса. Мы обсудим имеющиеся возможности C++ в главе 11).

Для вычисления значения функции operator+() мы сначала складываем значения feet и inches обоих операндов (корректируя результат, если это необходимо). Полученные значения f и i мы затем используем при инициализации безымянного объекта класса Distance, который затем будет возвращен в выражение.

```
return Distance ( f, i );
```

Это похоже на расположение, используемое в COUNTPP3, за исключением того, что конструктор принимает два аргумента вместо одного. В строке

```
dist3 = dist1 + dist2;
```

функции main() затем присваивается значение безымянного объекта класса Distance переменной dist3. Сравните это интуитивно-понятное выражение с использованием вызова функции для выполнения той же задачи, как это было сделано в программе ENGLCON из главы 6.

Похожие методы вы сможете создать в классе Distance для перегрузки других операций. Тогда вы сможете выполнять вычитание, умножение и деление с объектами этого класса, используя обычную запись выражений.

Объединение строк

Операция + не может быть использована для объединения строк. То есть мы не можем использовать выражение типа:

```
str3 = str1 + str2;
```

где str1, str2 и str3 — строковые переменные (массивы типа char), и «кот» плюс «бегемот» равно «котбегемот». Однако если мы будем использовать свой собственный класс String из программы STROBJ главы 6, то мы сможем перегрузить операцию + для объединения строк. Это же делает и стандартный класс string в C++, но будет проще пронаблюдать этот процесс на примере нашего более простого класса String. Перегрузка операции + для выполнения действий, которые являются не совсем сложением, будет являться нашим следующим примером переопределения языка C++. Приведем листинг программы STRPLUS.

```
// strplus.cpp
// перегрузка операции + для строк
#include <iostream>
using namespace std;
#include <string.h>   // для функций strcpy, strcat
#include <stdlib.h>   // для функции exit
///////////////////////////////////////////////////////////
class String          // наш класс для строк
{
  private:
    enum { SZ = 80; } // максимальный размер строки
    char str [ SZ ];  // массив для строки
  public:
    // конструктор без параметров
```

```
    String ( )
      { strcpy ( str, "" ); }
    // конструктор с одним параметром
    String ( char s [ ] )
      { strcpy ( str, s ); }
    // показ строки
    void display ( ) const
      { cout << str; }
    // оператор сложения
    String operator+ ( String ss ) const
      {
        String temp;  // временная переменная
        if ( strlen ( str ) + strlen ( ss.str ) < SZ )
        {
          strcpy ( temp.str, str );    // копируем содержимое первой строки
          strcat ( temp.str, ss.str ); // добавляем содержимое второй строки
        }
        else
        {
          cout << "\nПереполнение!";
          exit ( 1 );
        }
        return temp;  // возвращаем результат
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  String s1 = "\nС Рождеством! "; // используем конструктор с параметром
  String s2 = "С Новым годом!";   // используем конструктор с параметром
  String s3;                      // используем конструктор без параметров

  // показываем строки
  s1.display ( );
  s2.display ( );
  s3.display ( );

  s3 = s1 + s2;       // присваиваем строке s3 результат сложения строк s1 и s2

  s3.display ( );     // показываем результат
  cout << endl;

  return 0;
}
```

Сначала программа выводит на дисплей три различные строки (третья строка пуста, поэтому ничего не выведется). Далее первые две строки объединяются и помещаются в третью строку, которая затем будет выведена на экран. Вот результат работы программы:

```
С Рождеством! С Новым годом!        - s1, s2 и s3 (пока пуста)
С Рождеством! С Новым годом!        - s3 после сложения
```

Перегрузка операции + в этом случае похожа на то, что мы делали ранее. Объявление

```
String operator+ ( String ss );
```

показывает, что функция operator+() принимает один аргумент типа String и возвращает объект того же типа. При объединении строк с помощью функции operator+() создается временный объект типа String, в него копируется строка объекта, для которого вызвана функция. Затем, используя библиотечную функцию strcat(), мы присоединяем к ней строку из объекта аргумента и возвращаем полученную во временном объекте строку. Заметим, что мы не можем использовать

```
return String ( string );
```

такой подход, где создается безымянный временный объект типа String, так как нам нужен доступ к временному объекту не только для его инициализации, но и для присоединения к нему строки аргумента.

Вам нужно быть внимательными, чтобы не допустить переполнения длины строки, используемой в классе String. Для предупреждения такого случая мы в функции operator+() проверяем, не превышает ли общая длина двух объединяемых строк максимально возможную длину строки. Если превышает, то мы выводим сообщение об ошибке вместо выполнения операции объединения. (Вы можете сообщить об ошибке и другим путем, например возвратив программе 0 или исключив эту операцию, как мы обсудим в главе 14 «Шаблоны и исключения».)

Напомним, что при использовании enum для установки константы значение SZ временно фиксировано. Так как все компиляторы работают по стандартам C++, то вы можете изменить значение SZ

```
static const int SZ = 80;
```

Множественная перегрузка

Мы рассмотрели различные примеры использования операции +: для сложения интервалов и объединения строк. Мы можем использовать оба написанных нами класса в одной программе, и при этом компилятор будет знать, какая из функций нам нужна, так как он выбирает функцию, исходя из операндов операции.

Операции сравнения

Давайте рассмотрим перегрузку операций другого типа: операций сравнения.

Сравнение объектов класса Distance

В нашем первом примере мы перегрузим операцию «меньше чем» (<) в классе Distance для того, чтобы иметь возможность сравнивать объекты этого класса. Приведем листинг программы ENGLESS.

```
// engless.cpp
// перегрузка операции < для сравнения длин
#incude <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance // класс английских мер длины
{
  private:
```

```
    int feet;
    float inches;
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ), inches ( 0.0 )
      { }
    // конструктор с двумя параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in )
      { }
    // получение информации от пользователя
    void getdist ( )
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: ";  cin >> inches;
      }
    // показ информации
    void showdist ( )
      { cout << feet << "\'-" << inches << '\"'; }
    // сравнение двух длин
    bool operator< ( Distance ) const;
};
///////////////////////////////////////////////////////////
// сравнение двух длин
bool Distance::operator< ( Distance d2 ) const
{
  float bf1 = feet + inches / 12;
  float bf2 = d2.feet + d2.inches / 12;
  return ( bf1 < bf2 ) ? true : false;
}
///////////////////////////////////////////////////////////
int main ( )
{
  Distance dist1;              // определяем переменную
  dist1.getdist ( );           // получаем длину от пользователя

  Distance dist2 ( 6, 2.5 ); // определяем и инициализируем переменную

  // показываем длины
  cout << "\ndist1 = "; dist1.showdist ( );
  cout << "\ndist2 = "; dist2.showdist ( );

  // используем наш оператор
  if ( dist1 < dist2 )
    cout << "\ndist1 меньше чем dist2";
  else
    cout << "\ndist1 больше или равно чем dist2";
  cout << endl;

  return 0;
}
```

Эта программа сравнивает интервал, заданный пользователем, с интервалом 6'-2.5", определенным в программе. В зависимости от полученного результата затем выводится одно из двух предложений. Пример возможного вывода:

```
Введите футы: 5
Введите дюймы: 11.5
```

```
dist1 = 5'-11.5"
dist2 = 6'-2.5"
dist1 меньше чем dist2
```

Подход, используемый для функции operator<() в программе ENGLESS, похож на перегрузку операции + в программе ENGPLUS, за исключением того, что здесь функция operator<() имеет возвращаемое значение типа bool. Возвращаемым значением может быть false или true, в зависимости от результата сравнения двух интервалов. При сравнении мы конвертируем оба интервала в значения футов с плавающей точкой и затем сравниваем их, используя обычную операцию <. Помним, что здесь использована операция условия:

```
return ( bf1 < bf2 ) ? true : false;
```

это то же самое, что

```
if ( bf1 < bf2 )
  return true;
else
  return false;
```

Сравнение строк

Рассмотрим другой пример перегрузки оператора, теперь это будет оператор сравнения (==). Мы будем использовать его для сравнения объектов класса String, возвращая true, если они одинаковы, и false, если они различны. Листинг программы STREQUAL:

```
// strequal.cpp
// перегрузка операции ==
#include <iostream>
using namespace std;
#include <string.h> // для функции strcmp
///////////////////////////////////////////////////////////
class String             // наш класс для строк
{
  private:
    enum { SZ = 80; }    // максимальный размер строки
    char str [ SZ ];     // массив для строки
  public:
    // конструктор без параметров
    String ( )
      { strcpy ( str, "" ); }
    // конструктор с одним параметром
    String ( char s [ ] )
      { strcpy ( str, s ); }
    // показ строки
    void display ( ) const
      { cout << str; }
    // получение строки
    void getdist ( )
      { cin.get ( str, SZ ); }
    // оператор сравнения
    bool operator== ( String ss ) const
      {
        return ( strcmp ( str, ss.str ) == 0 ) ? true : false;
      }
};
```

```
///////////////////////////////////////////////////////////
int main ( )
{
  String s1 = "да";
  String s2 = "нет";
  String s3;

  cout << "\nВведите 'да' или 'нет': ";
  s3.getstr ( );        // получаем строку от пользователя

  if ( s3 == s1 )       // сравниваем со строкой 'да'
    cout << "Вы ввели 'да'\n";
  else
  if ( s3 == s2 )       // сравниваем со строкой 'нет'
    cout << "Вы ввели 'нет'\n";
  else
    cout << "Следуйте инструкциям!\n";

  return 0;
}
```

В функции main() мы используем операцию == дважды: в первый раз, когда определяем, введена ли строка «да», а во второй раз — введена ли строка «нет». Вот результат работы программы, когда пользователь печатает слово «да»:

```
Введите 'да' или 'нет': да
Вы ввели 'да'
```

Функция operator==() использует библиотечную функцию strcmp() для сравнения двух строк. Эта функция возвращает 0, если строки равны, отрицательное число, если первая строка меньше второй, и положительное число, если первая строка больше второй строки. Здесь «меньше чем» и «больше чем» использованы в их лексикографическом смысле для определения того, что стоит раньше в алфавитном списке — первая строка или вторая.

Другие операции сравнения, такие, как < и >, могут быть также использованы для сравнения лексикографического значения строк. Или, что то же самое, операторы сравнения могут быть переопределены для сравнения длины строк. Так как вы можете сами определять, для чего будет использоваться операция, то вы можете использовать любое подходящее для вашей ситуации определение.

Операции арифметического присваивания

Давайте закончим наше изучение перегрузки бинарных операций на примере арифметического присваивания: +=. Вспомним, что эта операция выполняет присваивание и сложение за один шаг. Мы будем использовать эту операцию для сложения интервалов, записывая результат в переменную, обозначающую первый интервал. Это похоже на пример ENGLPLUS, представленный нами ранее, но здесь есть небольшое отличие. Приведем листинг программы ENGLPLEQ.

```
// engpleq.cpp
// перегрузка операции +=
#include <iostream>
```

```
using namespace std;
///////////////////////////////////////////////////////////
class Distance          // класс английских мер длины
{
  private:
    int feet;
    float inches;
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ), inches ( 0.0 )
      { }
    // конструктор с двумя параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in )
      { }
    // получение информации от пользователя
    void getdist ( )
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: ";  cin >> inches;
      }
    // показ информации
    void showdist ( )
      { cout << feet << "\'-" << inches << '\"'; }
    // сложение с присвоением
    bool operator+= ( Distance ) const;
};
///////////////////////////////////////////////////////////
void Distance::operator+= ( Distance d2 )
{
  feet += d2.feet;       // складываем футы
  inches += d2.inches;   // складываем дюймы
  if ( inches >= 12.0 )  // если дюймов больше 12
  {
    inches -= 12.0;      // то уменьшаем дюймы на 12
    feet++;              // увеличиваем футы на 1
  }
}
///////////////////////////////////////////////////////////
int main ( )
{
  Distance dist1;               // определяем переменную
  dist1.getdist ( );            // и вводим информацию
  cout << "\ndist1 = "; dist1.showdist ( );

  Distance dist2 ( 11, 6.25 ); // описываем и инициализируем другую переменную
  cout << "\ndist2 = "; dist2.showdist ( );

  dist1 += dist2;               // dist1 = dist1 + dist2
  cout << "\nПосле добавления:";

  cout << "\ndist1 = "; dist1.showdist ( );
  cout << endl;

  return 0;
}
```

В этой программе мы получаем интервал от пользователя и складываем его со вторым интервалом, инициализированным в программе как 11'-6.25". Пример простого взаимодействия с программой:

```
Введите футы: 3
Введите дюймы: 5.75
dist1 = 3'-5.75"
dist2 = 11'-6.25"
После добавления:
dist1 = 15'-0"
```

В этой программе сложение выполняется в функции main() в строке

```
dist1 += dist2:
```

Это означает, что сумма интервалов dist1 и dist2 будет записана в переменную dist1.

Заметим, что существует различие между функцией operator+=(), используемой здесь, и функцией operator+() из программы ENGLPLUS. В функции operator+() создается новый объект типа Distance, который будет затем возвращен функцией, чтобы быть присвоенным третьему объекту типа Distance:

```
dist3 = dist1 + dist2;
```

В функции operator+=() программы ENGLPLEQ объектом, принимающим значение суммы, является объект, вызывающий функцию operator+=(). Поэтому feet и inches являются заданными величинами, а не временными переменными, используемыми только для возвращаемого объекта. Функция operator+=() не имеет возвращаемого значения: она возвращает тип void. Для операции += возвращаемое значение не требуется, так как мы не присваиваем ее результат никакой переменной. Эта операция используется без возвращаемого значения в выражениях, похожих на выражение из нашей программы:

```
dist1 += dist2;
```

Но если мы захотим использовать эту операцию в более сложных выражениях, таких, как:

```
dist3 = dist1 += dist2;
```

то в этом случае нам понадобится возвращаемое значение. Мы можем ввести его, записав в конце определения функции operator+=() строку

```
return Distance ( feet. inches );
```

в которой безымянный объект инициализируется тем же значением, что и объект, вызывающий функцию, и затем возвращается.

Операция индексации массива ([])

Операция индексации, которая обычно используется для доступа к элементам массива, может быть перегружена. Это полезно в том случае, если вы хотите изменить способ работы C++ с массивами. Например, если вам понадобится создать «безопасный массив», в котором заложена автоматическая проверка ис-

пользуемого для доступа к массиву индекса элемента. Она будет проверять, находитесь ли вы в границах массива. (Вы можете также использовать класс vector, описанный в главе 15 «Стандартная библиотека шаблонов (STL)».)

Для демонстрации перегрузки операции индексации массива мы должны вернуться к другой теме, с которой мы уже встречались в главе 5, — возвращению значений из функции по ссылке. Перегруженная операция индексации должна возвращать свои значения по ссылке. Чтобы показать, почему это так, мы рассмотрим три программы, которые реализуют «безопасный» массив, каждая при этом использует свой подход для вставки и считывания элементов массива:

- два отдельных метода put() и get();
- метод access(), использующий возвращение по ссылке;
- перегруженная операция [ ], использующая возвращение по ссылке.

Все три программы используют класс safearray, в котором определен массив, состоящий из 100 элементов типа int. И все три проверяют, находится ли элемент, к которому мы хотим получить доступ, в границах массива. Функция main() каждой из программ заполняет массив значениями (каждое из которых равно значению индекса массива, умноженному на 10) и затем выводит их все, чтобы показать, что пользователь сделал все правильно.

Два отдельных метода put() и get()

В первой программе мы используем для доступа к элементам массива два метода: putel() для вставки элемента в массив и getel() для получения значения нужного нам элемента. Оба метода проверяют, чтобы значение индекса массива входило в рамки границ массива. То есть оно не должно быть меньше 0 и больше размера массива (минус 1). Приведем листинг программы ARROVER1.

```
// arrover1.cpp
// демонстрация создания безопасного массива, проверяющего
// свои индексы при использовании
// используются отдельные функции для установки и получения значения
#include <iostream>
using namespace std;
#include <process.h>    // для функции exit
const int LIMIT = 100;  // размер массива
///////////////////////////////////////////////////////////
class safearray
{
  private:
    int arr [ LIMIT ];
  public:
    // установка значения элемента массива
    void putel ( int n, int elvalue )
      {
        if ( n < 0 || n >= LIMIT )
          { cout << "\nОшибочный индекс!"; exit ( 1 ); }
        arr [ n ] = elvalue;
      }
    // получение значения элемента массива
    int getel ( int n ) const
      {
```

```
      if ( n < 0 || n >= LIMIT )
        { cout << "\nОшибочный индекс!"; exit ( 1 ); }
      return arr [ n ];
    }
};
///////////////////////////////////////////////////////////
int main ( )
{
  safearray sa1;

  // задаем значения элементов
  for ( int j = 0; j < LIMIT; j++ )
    sa1.putel ( j, j * 10 );

  // показываем элементы
  for ( j = 0; j < LIMIT; j++ )
  {
    int temp = sa1.getel ( j );
    cout << "Элемент " << j << " равен " << temp << endl;
  }

  return 0;
}
```

Данные помещаются в массив с помощью метода putel() и выводятся на дисплей с помощью метода getel(). Безопасность массива реализована с помощью вывода сообщения об ошибке при попытке использования индекса, не входящего в границы массива. Однако этот формат несколько груб.

Метод access (), использующий возвращение по ссылке

Как оказывается, мы можем использовать один метод для вставки и вывода элементов массива. Секрет этого метода в использовании возвращения значения по ссылке. Это означает, что мы можем записать функцию с левой стороны знака равно, при этом переменной, стоящей справа, будет присвоено значение, возвращаемое функцией, как показано в главе 5. Приведем листинг программы ARROVER2.

```
// arrover2.cpp
// демонстрация создания безопасного массива, проверяющего
// свои индексы при использовании
// используется общая функция для установки и получения значения
#include <iostream>
using namespace std;
#include <process.h>    // для функции exit
const int LIMIT = 100;  // размер массива
///////////////////////////////////////////////////////////
class safearray
{
  private:
    int arr [ LIMIT ];
  public:
    // обратите внимание, что функция возвращает ссылку!
    int& access ( int n )
      {
        if ( n < 0 || n >= LIMIT )
          { cout << "\nОшибочный индекс!"; exit ( 1 ); }
```

```
        return arr [ n ];
      }
};
////////////////////////////////////////////////////////////
int main ( )
{
  safearray sa1;

  // задаем значения элементов
  for ( int j = 0; j < LIMIT; j++ )
    sa1.access ( j ) = j * 10; // используем функцию слева от знака =

  // показываем элементы
  for ( j = 0; j < LIMIT; j++ )
  {
    int temp = sa1.access ( j ); // используем функцию справа от знака =
    cout << "Элемент " << j << " равен " << temp << endl;
  }

  return 0;
}
```

Строка

```
sa1.access ( j ) = j * 10;
```

означает, что значение j*10 будет помещено в элемент массива arr[j], ссылка на который возвращается методом.

Это использование одного метода для ввода и вывода элементов массива в общем случае немного более удобно, чем использование отдельных методов, на одно имя меньше. Но существует еще лучший способ, вовсе без имен.

Перегруженная операция [], использующая возвращение по ссылке

Мы перегрузим операцию индексации [] в классе safearray для использования стандартной записи C++ для доступа к элементам массива. Однако, так как эта операция обычно используется слева от знака равно, наша перегруженная функция должна возвращать свое значение по ссылке, как показано в предыдущей программе. Листинг программы ARROVER3:

```
// arrover3.cpp
// демонстрация создания безопасного массива, проверяющего
// свои индексы при использовании
// используется перегрузка операции [ ]
#include <iostream>
using namespace std;
#include <process.h>       // для функции exit
const int LIMIT = 100;     // размер массива
////////////////////////////////////////////////////////////
class safearray
{
  private:
    int arr [ LIMIT ];
  public:
```

```
    // обратите внимание, что функция возвращает ссылку!
    int& operator[ ] ( int n )
      {
        if ( n < 0 || n >= LIMIT )
          { cout << "\nОшибочный индекс!"; exit ( 1 ); }
        return arr [ n ];
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  safearray sa1;

  // задаем значения элементов
  for ( int j = 0; j < LIMIT; j++ )
    sa1[ j ] = j * 10;    // используем функцию слева от знака =

  // показываем элементы
  for ( j = 0; j < LIMIT; j++ )
  {
    int temp = sa1 [ j ]; // используем функцию справа от знака =
    cout << "Элемент " << j << " равен " << temp << endl;
  }

  return 0;
}
```

В этой программе мы можем использовать обычную запись операции индексации массива

```
sa1 [ j ] = j * 10;
```

и

```
temp = sa1 [ j ];
```

для ввода и вывода элементов массива.

Преобразование типов

Вы уже знаете, что операция = используется для присваивания значения одной переменной другой в выражениях типа

```
intvar1 = intvar2;
```

где intvar1 и intvar2 — целые переменные. Вы могли также заметить, что операция = присваивает значение одного объекта, определенного пользователем, другому объекту того же типа, как в строке

```
dist3 = dist1 + dist2;
```

где результат операции сложения, имеющий тип Distance, присвоен другому объекту типа Distance — dist3. Обычно если значение одного объекта присваивается другому объекту того же типа, то значения всех переменных объекта просто копируются в новый объект. Компилятору не нужны специальные инструкции,

чтобы использовать операцию = для определенных пользователем типов, таких, как объекты класса Distance.

Таким образом, присвоение внутри типов, независимо от того, принадлежат они к основным или определенным пользователем типам, выполняется компилятором без усилий с нашей стороны, при этом с обеих сторон знака равно используются переменные одинаковых типов. Но что же случится, если с разных сторон знака равно будут стоять переменные разных типов? Это сложный вопрос, которому мы посвятим остаток этой главы. Сначала мы рассмотрим, как компилятор выполняет преобразование основных типов, которое происходит автоматически. Затем мы объясним несколько ситуаций, где компилятор не выполняет преобразование автоматически и мы должны определить, что он должен делать. В эти ситуации включены преобразования между основными и определенными пользователем типами и преобразования между различными определенными пользователем типами.

Может показаться, что это демонстрирует недостаточную практику программирования для конвертирования одного типа в другой. Ведь в таких языках, как Pascal, предприняты попытки избавить вас от проделывания подобных преобразований. Однако философия C++ (и C) — это гибкость, предоставленная возможностью преобразования, которая перевешивает недостатки. Это спорный вопрос, и мы вернемся к нему в конце этой главы.

Преобразования основных типов в основные типы

Когда мы пишем строку типа

```
intvar = floatvar;
```

где intvar — переменная типа int, а floatvar — переменная типа float, то мы предполагаем, что компилятор вызовет специальную функцию для преобразования значения переменной floatvar, которая имеет формат числа с плавающей точкой, в формат целого числа, чтобы мы могли присвоить его переменной intvar. Конечно, существует множество таких преобразований: из float в double, из char во float и т. д. Каждое такое преобразование имеет свою процедуру, встроенную в компилятор, которая вызывается в зависимости от типов переменных, стоящих с обеих сторон знака равно. Мы говорим, что преобразования *неявные*, так как они не отражены в листинге.

Иногда мы хотим принудительно заставить компилятор преобразовать один тип в другой. Для этого используется оператор cast. Например, для преобразования float в int мы можем записать следующее выражение:

```
intvar = static_cast<int>( floatvar );
```

Использование этого оператора предоставляет нам *явное* преобразование: очевидно, что в листинге функция static_cast<int>() предназначена для преобразования float в int. Однако такое явное преобразование использует те же встроенные процедуры, что и неявное.

Преобразования объектов в основные типы и наоборот

Если нам нужно осуществить преобразование определенных пользователем типов в основные типы, то мы не можем положиться на встроенные функции, так как компилятору не известно об определенных пользователем типах ничего, кроме того, что мы скажем ему сами. Поэтому мы должны сами написать функции для преобразования типов.

В нашем следующем примере показано, как происходит преобразование основного типа в тип, определенный пользователем. В этом примере в качестве определенного пользователем типа мы будем использовать класс **Distance** из предыдущих примеров, а в качестве основного типа — тип **float**, который мы использовали для переменной, содержащей метры.

В примере показаны оба варианта преобразования: из **Distance** в **float** и из **float** в **Distance**. Листинг программы **ENGLCONV**:

```
// englconv.cpp
// перевод длины из класса Distance в метры и обратно
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance          // класс английских мер длины
{
  private:
    const float MTF;    // коэффициент перевода метров в футы
    int feet;
    float inches;
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ), inches ( 0.0 ), MTF ( 3.280833F )
      { }
    // конструктор с одним параметром,
    // переводящий метры в футы и дюймы
    Distance ( float meters ) : MTF ( 3.280833F )
      {
        float fltfeet = MTF * meters;      // переводим в футы
        feet = int ( fltfeet );            // берем число полных футов
        inches = 12 * ( fltfeet - feet ); // остаток - это дюймы
      }
    // конструктор с двумя параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in ), MTF ( 3.280833F )
      { }
    // получение информации от пользователя
    void getdist ( )
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: ";  cin >> inches;
      }
    // показ информации
    void showdist ( ) const
      { cout << feet << "\'-" << inches << '\"' }
    // оператор перевода для получения метров из футов
    operator float ( ) const
      {
```

```
        float fracfeet = inches / 12;          // переводим дюймы в футы
        fracfeet += static_cast<float>( feet ); // добавляем целые футы
        return fracfeet / MTF;                 // переводим в метры
      }
};
//////////////////////////////////////////////////////////
int main ( )
{
  float mtrs;
  Distance distl = 2.35F; // используется конструктор, переводящий метры в футы и дюймы

  cout << "\ndistl = "; distl.showdist ( );

  mtrs = static_cast<flost>( distl ); // используем оператор перевода в метры

  cout << "\ndistl = " << mtrs << " meters\n";

  Distance dist2 ( 5, 10.25 ); // используем конструктор с двумя параметрами

  mtrs = dist2; // неявно используем перевод типа
  cout << "\ndist2 = " << mtrs << " meters\n";

  // dist2 = mtrs; // а вот это ошибка - так делать нельзя
  return 0;
}
```

В функции main() сначала происходит преобразование фиксированной величины типа float — 2.35, представляющей собой метры, — в футы и дюймы с использованием конструктора с одним аргументом:

```
Distance distl = 2.35F;
```

Затем мы преобразовываем тип Distance во float в строке

```
mtrs = static_cast<float>( dist2 );
```

и

```
mtrs = dist2;
```

Результат работы программы:

```
distl = 7'-8.51949"- это то же, что и 2.35 метра
distl = 2.35 meters- это то же, что и 7'-5.51949"
dist2 = 1.78435 meters- это то же, что и 5'-10.25"
```

Мы увидели, как в функции main() выполняется преобразование типов с использованием простого выражения присваивания. Теперь рассмотрим процесс изнутри, в функциях класса Distance. Преобразование определенного пользователем типа в основной требует другого подхода, нежели преобразование основного типа в определенный пользователем. Мы рассмотрим, как оба типа преобразований выполняются в программе ENGLCONV.

От основного к определенному пользователем

Для перехода от основного типа — в нашем случае float — к определенному пользователем типу, такому, как Distance, мы используем конструктор с одним аргу-

ментом. Его иногда называют конструктором преобразования. Вот как этот конструктор выглядит в программе ENGLCONV:

```
Distance ( float meters )
{
  float fltfeet = MTF * meters;
  feet = int ( fltfeet );
  inches = 12 * ( fltfeet - feet );
}
```

Этот конструктор вызывается, когда создается объект класса Distance с одним аргументом. Конструктор предполагает, что аргумент представляет собой метры. Он преобразовывает аргумент в футы и дюймы и присваивает полученное значение объекту. Таким образом, преобразование от метров к переменной типа Distance выполняется вместе с созданием объекта в строке

```
Distance dist1 = 2.35;
```

От определенного пользователем к основному

А как же насчет преобразования от определенного пользователем типа к основному? Здесь вся хитрость в том, чтобы создать что-то, называемое операцией преобразования. Вот как мы делаем это в программе ENGLCONV:

```
operator float ( )
{
  float fracfeet = inches / 12;
  fracfeet += float ( feet );
  return fracfeet / MTF;
}
```

Эта операция принимает значение объекта класса Distance, преобразовывает его в значение типа float, представляющее собой метры, и возвращает это значение.

Операция может быть вызвана явно

```
mtrs = static_cast<float>( dist1 );
```

или с помощью простого присваивания

```
mtrs = dist2;
```

В обоих случаях происходит преобразование объекта типа Distance в его эквивалентное значение типа float, представляющее собой метры.

Преобразования строк в объекты класса string и наоборот

Рассмотрим другой пример, в котором использованы конструктор с одним аргументом и оператор преобразования. Здесь используется класс String, который мы видели в примере STRPLUS ранее в этой главе.

```
// strconv.cpp
// перевод обычных строк в класс String
#include <iostream>
using namespace std;
#include <string.h>          // для функций str*
```

```
/////////////////////////////////////////////////////////////
class String
{
  private:
    enum { SZ = 80; }          // размер массива
    char str [ SZ ];           // массив для хранения строки
  public:
    // конструктор без параметров
    String ( )
      { str [ 0 ] = '\x0'; }
    // конструктор с одним параметром
    String ( char s [ ] )
      { strcpy ( str, s ); } // сохраняем строку в массиве
    // показ строки
    void display ( ) const
      { cout << str; }
    // перевод строки к обычному типу
    operator char* ( )
      { return str; }
};
/////////////////////////////////////////////////////////////
int main ( )
{
  String s1;                  // используем конструктор без параметров

  char xstr [ ] = "Ура, товарищи! "; // создаем обычную строку

  s1 = xstr;                  // неявно используем конструктор с одним параметром

  s1.display ( );             // показываем строку

  String s2 = "Мы победим!"; // снова используем конструктор с параметром

  cout << static_cast<char*>( s2 ); // используем оператор перевода типа
  cout << endl;

  return 0;
}
```

Конструктор с одним аргументом преобразовывает обыкновенную строку (массив элементов типа char) в объект класса String:

```
String ( char s [ ] )
  { strcpy ( str, s ); }
```

Строка передается в качестве аргумента и копируется в переменную класса str заново созданного объекта класса String, используя библиотечную функцию strcpy().

Это преобразование будет применено, когда создается объект класса String

```
String s2 = "Мы победим!";
```

или будет применено в выражении присваивания

```
s1 = xstr;
```

где s1 — объект класса String, а переменная xstr — просто строка.

Операция преобразования используется для перевода объекта класса String в строку:

```
operator char* ( )
  { return str; }
```

Звездочка в этом выражении означает *указатель на*. Мы не будем вдаваться в объяснения указателей до главы 10, но их использование нетрудно понять. Это означает указатель на char, что почти то же, что и массив типа char. То есть запись char* — почти то же, что char[]. Это просто другой способ объявления переменной строкового типа.

Оператор преобразования используется компилятором в строке

```
cout << static_cast<char*>( s2 );
```

Здесь s2 — переменная, использованная в качестве аргумента для перегруженной операции <<. Так как операции << ничего не известно об определенном пользователем типе String, то компилятор будет пытаться преобразовать переменную s2 к одному из известных ему типов. Мы определяем тип, к которому хотим преобразовать переменную, с помощью оператора static_cast для char*. Оператор ищет преобразование от типа String к строковому типу и, найдя нашу функцию char*(), использует ее для генерации строки, которая затем будет выведена через операцию <<. (Результат похож на вызов метода String::display(), но делает легким и интуитивно понятным вывод с помощью операции <<, а метод display() будет уже лишним и может быть удален.)

Вот результат работы программы STRCONV:

```
Ура, товарищи! Мы победим!
```

Пример STRCONV демонстрирует, что преобразование типов происходит автоматически не только в выражениях присваивания, но и в других подходящих местах, таких, как пересылка аргументов операциям (например, <<) или функциям. При передаче в операцию или функцию аргумента неправильного типа, они преобразуют его к приемлемому типу, если вы определили такое преобразование.

Заметим, что вы не можете использовать явное выражение присваивания для преобразования объекта класса String к переменной строкового типа:

```
xstr = s2;
```

Строка xstr — это массив, и вы не можете ничего ему присвоить (хотя, как мы увидим в главе 11, если мы перегрузим операцию присваивания, то все станет возможным).

Преобразования объектов классов в объекты других классов

Что мы можем сказать о преобразованиях между объектами различных, определенных пользователем классов? Для них применяются те же два метода преобразования, что и для преобразований между основными типами и объектами опре-

деленных пользователем классов. То есть вы можете использовать конструктор с одним аргументом или операцию преобразования. Выбор зависит от того, хотите ли вы записать функцию преобразования в классе для исходного объекта или для объекта назначения. Например, предположим, что мы записали:

```
objecta = objectb;
```

где objecta — объект класса A, а objectb — объект класса B. Где же расположить функцию преобразования, в классе A (это класс назначения, так как objecta получает значение) или в классе B? Мы рассмотрим оба варианта.

Две формы времяисчисления

В нашей программе мы будем производить преобразования между двумя способами исчисления времени: 12-часовым и 24-часовым. Эти способы еще называют гражданским временем и военным временем. Класс time12 будет предназначен для гражданского времени, которое используется в цифровых часах и табло отправки и прибытия самолетов. Мы предполагаем, что нам не нужны секунды, поэтому в классе time12 мы будем использовать только часы (от 1 до 12) и минуты, а также обозначение времени суток «a.m.» и «p.m.», что означает *до полудня* и *после полудня* соответственно. Класс time24, предназначенный для более точных целей, таких, как воздушная навигация, использует часы (от 00 до 23), минуты и секунды. В табл. 8.1 показаны различия между двумя формами времяисчисления.

Таблица 8.1. 12-часовое и 24-часовое время

12-часовое время	24-часовое время
12:00 a.m. (полночь)	00:00
12:01 a.m.	00:01
1:00 a.m.	01:00
6:00 a.m.	06:00
11:59 a.m.	11:59
12:00 p.m. (полдень)	12:00
12:01 p.m.	12:01
6:00 p.m.	18:00
11:59 p.m.	23:59

Заметим, что 12 часов ночи (полночь) в гражданском времени соответствуют 00 часам в военном времени. В гражданском времени 0 часов нет.

Функция в исходном объекте

В нашей первой программе мы рассмотрим случай, когда функция преобразования расположена в исходном классе. В этом случае она будет реализована в виде операции преобразования. Приведем листинг программы TIMES1.

```
// times1.cpp
// программа перевода времени в 24-часовом написании
// в 12-часовое
```

```
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class time12
{
  private:
    bool pm;
    int hrs;
    int mins;
  public:
    time12 ( ) : pm ( true ), hrs ( 0 ), mins ( 0 )
      { }
    time12 ( bool ap, int h, int m ) : pm ( ap ), hrs ( h ), mins ( m )
      { }
    void display ( ) const
      {
        cout << hrs << ':';
        if ( mins < 10 )
          cout << '0';
        cout << mins << ' ';
        string am_pm = pm ? "p.m." : "a.m.";
        cout << am_pm;
      }
};
///////////////////////////////////////////////////////////
class time24
{
  private:
    int hours;
    int minutes;
    int seconds;
  public:
    time24 ( ) : hours ( 0 ), minutes ( 0 ), seconds ( 0 )
      { }
    time24 ( int h, int m, int s ) : hours ( h ), minutes ( m ), seconds ( s )
      { }
    void display ( ) const
      {
        if ( hours < 10 ) cout << '0';
        cout << hours << ':';
        if ( minutes < 10 ) cout << '0';
        cout << minutes << ':';
        if ( seconds < 10 ) cout << '0';
        cout << seconds;
      }
    operator time12 ( ) const;
};
///////////////////////////////////////////////////////////
time24::operator time12 ( ) const
{
  int hrs24 = hours;
  bool pm = hours < 12 ? false : true;

  int roundMins = seconds < 30 ? minutes : minutes + 1;
  if ( roundMins == 60 )
  {
```

```
    roundMins = 0;
    ++hrs24;
    if ( hrs24 == 12 ||hrs24 == 24 )
      pm = ( pm == true ) ? false : true;
  }
  int hrs12 = ( hrs24 < 13 ) ? hrs24 : hrs24 - 12;
  if ( hrs12 == 0 )
  {
    hrs12 = 12;
    pm = false;
  }
  return time12 ( pm, hrs12, roundMins );
}
///////////////////////////////////////////////////////////
int main ( )
{
  int h, m, s;

  while ( true )
  {
    cout << "Введите время в 24-часовом формате: \n";
    cout << "  Часы (от 0 до 23): "; cin >> h;
    if ( h > 23 )
      return ( 1 );
    cout << "  Минуты: "; cin >> m;
    cout << "  Секунды: "; cin >> s;

    time24 t24 ( h, m, s );
    cout << "Исходное время: ";
    t24.display ( );

    time12 t12 = t24;

    cout << "\nВ 12-часовом формате: ";
    t12.display ( );
    cout << "\n\n";
  }
  return 0;
}
```

В функции main() программы TIMES1 мы определили объект класса time24, названный t24, и присвоили ему значения часов, минут и секунд, полученные от пользователя. Мы также определили объект класса time12, названный t12, и инициализировали его значением объекта t24 в строке

```
time 12 t12 = t24;
```

Так как это объекты различных классов, то для осуществления присваивания необходимо преобразование. Мы определили здесь операцию преобразования в классе time24. Вот ее объявление:

```
time24::operator time12 ( ) const;
```

Эта функция преобразует вызывающий ее объект в объект класса time12 и возвращает функции main() объект, который будет затем присвоен переменной t12. Пример взаимодействия с программой TIMES1:

```
Введите время в 24-часовом формате:
  Часы (от 0 до 23): 17
  Минуты: 59
  Секунды: 45
Исходное время: 17:59:45
В 12-часовом формате: 6:00 p.m.
```

Значение секунд округляется, и мы получим 6:00 p.m. При попытке ввода значения часа больше 23 программа завершится.

Функция в объекте назначения

Давайте рассмотрим, как это же преобразование выполняется, если функция преобразования находится в классе назначения. В этой ситуации используется конструктор с одним аргументом. Однако все усложняется тем фактом, что конструктор класса назначения должен иметь доступ к данным исходного класса для выполнения преобразования. Поля класса time24 — hours, minutes и seconds — объявлены как private, поэтому мы должны написать специальные методы в классе time24, которые будут позволять к ним прямой доступ. Это методы getHrs(), getMins() и getSecs().

Листинг программы TIMES2:

```
// times2.cpp
// преобразование времени с использованием конструктора
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class time24
{
  private:
    int hours;
    int minutes;
    int seconds;
  public:
    time24 ( ) : hours ( 0 ), minutes ( 0 ), seconds ( 0 )
      { }
    time24 ( int h, int m, int s ) : hours ( h ), minutes ( m ), seconds ( s )
      { }
    void display ( ) const
      {
        if ( hours < 10 ) cout << '0';
        cout << hours << ':';
        if ( minutes < 10 ) cout << '0';
        cout << minutes << ':';
        if ( seconds < 10 ) cout << '0';
        cout << seconds;
      }
    int getHrs ( ) const;
    int getMins ( ) const;
    int getSecs ( ) const;
};
////////////////////////////////////////////////////////////
class time12
{
```

```
  private:
    bool pm;
    int hrs;
    int mins;
  public:
    time12 ( ) : pm ( true ), hrs ( 0 ), mins ( 0 )
      { }
    time12 ( time24 );
    time12 ( bool ap, int h, int m ) : pm ( ap ), hrs ( h ), mins ( m )
      { }
    void display ( ) const
      {
        cout << hrs << ':';
        if ( mins < 10 ) cout << '0';
        cout << mins << ' ';
        string am_pm = pm ? "p.m." : "a.m.";
        cout << am_pm;
      }
};
///////////////////////////////////////////////////////////
time12::time12 ( time24 t24 )
{
  int hrs24 = t24.getHrs ( );

  pm = t24.getHrs ( ) < 12 ? false : true;

  mins = ( t24.getSecs ( ) < 30 ) ? t24.getMins ( ) : t24.getMins ( ) + 1;
  if ( mins == 60 )
  {
    mins = 0;
    ++hrs24;
    if ( hrs24 == 12 || hrs24 == 24 )
      pm = ( pm == true ) ? false : true;
  }
  hrs = ( hrs24 < 13 ) ? hrs24 : hrs24 - 12;
  if ( hrs == 0 )
    { hrs = 12; pm = false; }
}
///////////////////////////////////////////////////////////
int main ( )
{
  int h, m, s;

  while ( true )
  {
    cout << "Введите время в 24-часовом формате: \n";
    cout << "  Часы (от 0 до 23): "; cin >> h;
    if ( h > 23 )
      return ( 1 );
    cout << "  Минуты: "; cin >> m;
    cout << "  Секунды: "; cin >> s;

    time24 t24 ( h, m, s );
    cout << "Исходное время: ";
    t24.display ( );

    time12 t12 = t24;
```

```
    cout << "\nВ 12-часовом формате: ";
    t12.display ( );
    cout << "\n\n";
  }
  return 0;
}
```

Функцией преобразования здесь является конструктор с одним аргументом из класса time12. Она устанавливает для объекта, ее вызвавшего, значение, соответствующее значению объекта класса time24, полученному в качестве аргумента. Это работает во многом так же, как и операция преобразования в программе TIMES1, за исключением того, что здесь возникает трудность с доступом к полям объекта класса time24, и нам необходимо использовать методы getHrs() и ему подобные.

Функция main() программы TIMES2 такая же, как в программе TIMES1. В этой строке вновь происходит преобразование типов от time24 к time12, но уже с использованием конструктора с одним аргументом:

```
time12 t12 = t24;
```

Результат работы программы будет таким же. Все различия находятся в коде, где предпочтение при выполнении отдается конструктору объекта назначения, а не операции преобразования исходного объекта.

Преобразования: когда что использовать

Когда же нам использовать конструктор с одним аргументом из класса назначения, а когда операцию преобразования из исходного класса? Обычно у вас есть возможность сделать выбор. Однако иногда выбор уже сделан за вас. Если вы покупаете библиотеку классов, то вы можете не иметь доступа к ее исходным файлам. Если вы будете использовать объект такого класса в качестве исходного в преобразовании, то вы сможете получить доступ только к классу назначения, поэтому вам нужно будет использовать конструктор с одним аргументом. Если же объект назначения принадлежит библиотечному классу, то вы должны использовать операцию преобразования в исходном классе.

Диаграммы классов UML

Мы познакомились с UML в главе 1 «Общие сведения». Теперь, когда мы уже немного знаем о классах, давайте рассмотрим наш первый пример использования UML: *диаграмму классов*. Эта диаграмма предлагает новый способ для отражения объектно-ориентированной программы и может пролить немного света на работу программ TIMES1 и TIMES2.

Взглянув на листинг программы TIMES1, мы увидим в нем два класса: time12 и time24. В диаграммах UML классы представлены в виде прямоугольников, как показано на рис. 8.3.

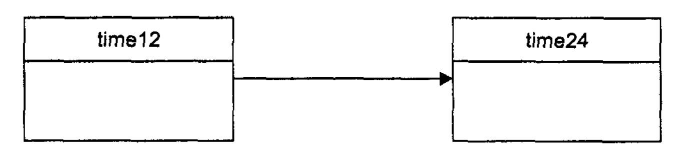


Рис. 8.3. Диаграмма классов UML для программы TIMES1

Прямоугольник каждого из классов разделен на секции горизонтальными линиями. Имя класса расположено в верхней секции. Здесь мы этого не сделали, но вы можете сами добавить в каждый класс секции для полей класса (называемых в UML *атрибутами*) и методов класса (называемых *операциями*).

Объединения

Классы могу иметь различные виды взаимоотношений. Классы программы TIMES1 образуют объединение. Мы обозначили его линией, соединяющей прямоугольники классов. (В главе 9 «Наследование» мы рассмотрим другой вид взаимоотношений — обобщение.)

Что же входит в объединение? Объекты реального мира, которые представлены в программе в виде классов, имеют между собой некоторые очевидные связи. Водители связаны с машинами, книги — с библиотеками, лошадиные скачки — с ипподромом. Если бы эти объекты были классами программы, то они входили бы в объединение.

В программе TIMES2 класс time12 объединен с классом time24, потому что мы преобразовываем объекты одного класса в объекты другого.

Объединение классов подразумевает, что объекты классов, в большей степени, чем сами классы, имеют взаимосвязи. Обычно два класса объединены, если объект одного из классов вызывает метод (операцию) объекта другого класса. Объединение возникает также, если атрибут одного из классов является объектом другого класса.

В программе TIMES1 объект t12 класса time12 вызывает функцию преобразования time12() объекта t24 из класса time24. Это происходит в функции main() в строке

```
time12 t12 = t24;
```

Этот вызов отражен на диаграмме линией, связывающей два класса.

Направленность

На диаграмме мы можем нарисовать незакрашенную стрелку, показывающую направленность объединения (как мы увидим позже, закрашенная стрелка имеет другое значение). Так как класс time12 вызывает класс time24, то стрелка будет показывать в направлении от класса time12 к классу time24. Мы назовем это объединение однонаправленным, так как действие происходит в одном направлении. Если каждый из двух классов вызывает операции другого класса, то такое объединение будет называться двунаправленным и будет обозначаться стрелками на обоих концах линии, соединяющей классы. Стрелки, обозначающие направленность, не являются обязательными, как и многие другие вещи в UML.

«Подводные камни» перегрузки операций и преобразования типов

Перегрузка операций и преобразование типов дают нам возможность создать, в сущности, новый язык. Пусть a, b и c — объекты определенного пользователем класса, а операция + перегружена, при этом строка

```
a = b + c;
```

может означать что-либо отличное от того, что было бы, если бы переменные a, b и c принадлежали одному из основных типов. Возможность переопределения встроенных блоков языка хороша тем, что вы можете сделать листинг программы более понятным и читаемым. Но возможен и другой результат, когда ваш листинг станет более непонятным и сложным. Приведем некоторые руководящие указания по этому поводу.

Использование похожих значений

Используйте перегрузку операций для выполнения действий, которые можно выполнить с помощью основных типов данных. Например, вы можете перегрузить знак + для выполнения вычитания, но едва ли это сделает ваш листинг более понятным.

Перегрузка операции предполагает, что это имеет смысл для выполнения определенных действий с объектами определенного класса. Если мы собираемся перегрузить операцию + класса X, то результат сложения двух объектов класса X должен иметь значение, по крайней мере, похожее на сумму. Например, в этой главе мы показывали, как перегрузить операцию + для класса Distance. Сложение двух интервалов несомненно имеет смысл. Мы также перегружали операцию + для класса String. Здесь мы интерпретировали сложение двух строк как добавление одной строки в конец другой для формирования третьей. Этот случай также удовлетворяет нашим требованиям. Но для многих классов, возможно, просто не будет причины говорить о «складывании» их объектов. Так, нам не нужно сложение для объектов класса employee, содержащих информацию о служащем.

Использование похожего синтаксиса

Используйте перегруженные операции так же, как и основные. Например, если alpha и beta основного типа, то операция присваивания в выражении

```
alpha += beta;
```

присваивает alpha сумму alpha и beta. Любая перегруженная версия этой операции должна выполнять что-либо подобное. Она, возможно, должна делать то же, что и

```
alpha = alpha + beta;
```

где + перегружен.

Если вы перегружаете одну арифметическую операцию, то вы можете для целостности перегрузить их все. Это предотвратит путаницу.

Некоторые синтаксические характеристики операций не могут быть изменены. Как вы могли заметить, вы не можете перегрузить бинарную операцию в унарную и наоборот.

Показывайте ограничение

Помните, что если вы перегрузили операцию +, то любому человеку, который не знаком с вашим листингом, нужно будет многое изучить, чтобы понять, что выражение

```
a = b + c;
```

в действительности означает. Если у вас очень много перегруженных операций, и они используются для не понятных интуитивно целей, то потеряется суть их использования и чтение листинга станет тяжелее вместо того, чтобы стать легче. Используйте перегрузку операций очень аккуратно, только там, где необходимость ее использования очевидна. Вас могут одолеть сомнения, использовать ли функцию взамен перегруженной операции, так как имя функции однозначно. Если вы, скажем, напишете функцию для поиска левой стороны строки, то вам будет лучше вызвать функцию getleft(), чем пытаться перегрузить операцию (например, &&) для тех же целей.

Избегайте неопределенности

Предположим, что вы используете и конструктор с одним аргументом, и операцию преобразования для выполнения некоторого преобразования (например, time24 в time12). Как компилятор узнает, какое из преобразований выполнять? Никак. Компилятор не следует вовлекать в ситуации, в которых ему не известно, что следует делать, так как он выдаст ошибку. Поэтому избегайте выполнения одного преобразования с помощью нескольких разных способов.

Не все операции могут быть перегружены

Следующие операции не могут быть перегружены: операция доступа к членам структуры или класса (.), операция разрешения (::) и операция условия (?:). А также операция (->), с которой мы еще не сталкивались. Кроме того, нельзя создавать новые операции (например, нельзя определить новую операцию возведения в степень **, которая есть в некоторых языках) и пытаться их перегрузить; перегружать можно только существующие операции.

Ключевые слова explicit и mutable

Давайте посмотрим на два необычных слова: explicit и mutable. Их использование дает различные эффекты, но мы сгруппировали их вместе, потому что они оба предназначены для изменения членов класса. Ключевое слово explicit относится к преобразованию типов, а mutable предназначено для более хитрых целей.

Предотвращение преобразования типов с помощью explicit

Преобразование, которое вы решили осуществить, может быть каким-то очень специальным. И вы начинаете его осуществлять, устанавливая подходящие операции преобразования и конструкторы с одним аргументом, как это показано в примерах TIMES1 и TIMES2. Однако могут появляться другие преобразования, которые вы не хотите делать. И вы активно препятствуете их осуществлению, чтобы предотвратить неприятные сюрпризы.

Легко предотвратить выполнение операции преобразования: вы можете просто не определять этой операции. Но с конструкторами не все так просто. Вам может понадобиться сконструировать объекты с использованием одного значения другого типа, при этом вы не хотите, чтобы осуществилось скрытое преобразование с помощью конструктора с одним аргументом. Что же делать?

В стандарт языка C++ включено ключевое слово explicit, предназначенное для решения этой проблемы. Оно просто помещается перед объявлением конструктора с одним аргументом. В программе EXPLICIT (основанной на программе ENGLCON) показано, как это выглядит.

```
// explicit.cpp
#include <iostream>
using namespace std:
///////////////////////////////////////////////////////////
class Distance
{
  private:
    const float MTF:
    int feet:
    float inches:
  public:
    Distance ( ) : feet ( 0 ). inches ( 0.0 ). MTF ( 3.280833F )
      { }
    explicit Distance ( float meters ) : MTF ( 3.280833F )
      {
        float fltfeet = MTF * meters:
        feet = int ( fltfeet ):
        inches = 12 * ( fltfeet - feet ):
      }
    void showdist ( )
      { cout << feet << "\'-" << inches << '\"': }
}:
///////////////////////////////////////////////////////////
int main ( )
{
  void fancyDist ( Distance ):
  Distance ( dist1 ( 2.35F ):

  // Distance dist1 = 2.35F:
  cout << "\ndist1 = ": dist1.showdist ( ):

  float mtrs = 3.0F:
  cout << "\nDist1 ":
  // fancyDist ( mtrs ):
```

```
  return 0;
}
//////////////////////////////////////////////////////////
void fancyDist ( Distance d )
{
  cout << "(в футах и дюймах) = ";
  d.showdist ( );
  cout << endl;
}
```

Эта программа включает в себя функцию fancyDist(), которая оформляет вывод объекта класса Distance, печатая фразу «в футах и дюймах» перед цифрами значений футов и дюймов. Аргументом этой функции является переменная класса Distance, и вы можете вызвать функцию fancyDist() с такой переменной без проблем.

Хитрость этой функции в том, что вы можете также вызывать функцию fancyDist() с переменной типа float в качестве аргумента:

```
fancyDist ( mtrs );
```

Компилятор поймет, что тип аргумента неверен и будет искать операцию преобразования. Найдя конструктор Distance, который принимает в качестве аргумента переменную типа float, компилятор приспособит этот конструктор для преобразования типа float в Distance и передаст значение типа Distance в функцию. Это неявное преобразование, одно из тех, которые вы можете упустить из вида.

Однако если мы сделаем конструктор явным, то мы предупредим неявные преобразования. Вы можете проверить это, удалив символ комментария из вызова функции fancyDist() в программе: компилятор сообщит вам о том, что не может выполнить преобразование. Без ключевого слова explicit этот вызов работает.

Отметим такой побочный эффект явного конструктора, как то, что вы не можете использовать инициализацию объекта, в которой присутствует знак равенства

```
Distance dist1 = 2.35F;
```

тогда как выражение со скобками

```
Distance dist1 ( 2.35F );
```

работает как обычно.

Изменение данных объекта, объявленных как const, используя ключевое слово mutable

Обычно при создании объекта-константы (как описано в главе 6) вам нужна гарантия того, что данные объекта невозможно будет изменить. Однако иногда случаются ситуации, когда вы хотите создать объект-константу, имеющий определенное поле, которое нужно будет изменять, несмотря на то, что сам объект является константой.

В качестве примера давайте представим окно (которое открывается на экране для программы). Некоторые возможности окна, такие, как полосы прокрутки

и меню, могут принадлежать окну. Принадлежность обычна для различных программных ситуаций и отличается большей степенью независимости, чем когда один объект является атрибутом другого. В таких ситуациях объект может оставаться неизменным, за исключением того, что изменяет его владелец. Полосы прокрутки сохраняют тот же размер, цвет и ориентацию, но собственность на них может передаваться от одного окна к другому. Это похоже на ситуацию, которая случается, когда ваш банк продаст вашу закладную другому банку; все условия заклада те же, но собственник другой.

Предположим, мы хотим иметь возможность создать константные полосы прокрутки, все атрибуты которых остаются неизменными, за исключением их владельца. Тогда нам и пригодится ключевое слово mutable. В программе MUTABLE показано, как это выглядит.

```
// mutable.cpp
#include <iostream>
#include <string>
using namespace std;
//////////////////////////////////////////////////////////
class scrollbar
{
  private:
    int size;
    mutable string owner;
  public:
    scrollbar ( int sz, string own ) : size ( sz ), owner ( own )
      { }
    void setSize ( int sz )
      { size = sz; }
    void setOwner ( string own ) const
      { owner = own; }
    int getSize ( ) const
      { return size; }
    string getOwner ( ) const
      { return owner; }
};
//////////////////////////////////////////////////////////
int main ( )
{
  const scrollbar sbar ( 60, "Приложение 1" );

  // sbar.setSize ( 100 );
  sbar.setOwner ( "Приложение 2" );

  cout << sbar.getSize ( ) << ", " << sbar.getOwner ( ) << endl;

  return 0;
}
```

Атрибут size из данных объекта класса scrollbar не может быть изменен в объекте-константе. Однако атрибут owner может быть изменен даже если объект — константа. Для того чтобы позволить это, используем ключевое слово mutable. В функции main() мы создаем объект-константу sbar. Его размер не может быть модифицирован, разве что владельцем с использованием метода setOwner(). (Если объект не константа, то, конечно, можно модифицировать оба атрибута.)

В этой ситуации считается, что sbar является логически константой. Это означает, что в теории он не может быть модифицирован, но на практике в некоторых случаях модифицируется.

Резюме

В этой главе мы увидели, как обычным операциям языка C++ придать новый смысл при использовании их с определенными пользователем типами. Ключевое слово operator используется для перегрузки операций, при этом операция будет выполнять действия, которые определит для нее программист.

Преобразование типов близко к перегрузке операций. Некоторые преобразования происходят между основными типами и типами, определенными пользователем. При этом используются два подхода: использование конструктора с одним аргументом при изменении основного типа на определенный пользователем и использование операции преобразования при изменении определенного пользователем типа на основной. Если преобразования происходят между различными определенными пользователем типами, то используются оба этих подхода.

В табл. 8.2 отражены возможные варианты преобразований.

Таблица 8.2. Преобразование типов

-	Процедура в классе назначения	Процедура в исходном классе
Основной в основной	(Встроенные операции преобразования)	
Основной к классу	Конструктор	N/A
Класс в основной	N/A	Операция преобразования
Класс в класс	Конструктор	Операция преобразования

Конструктор, объявленный с ключевым словом explicit, не может быть использован в ситуации неявного преобразования данных. Данные, объявленные с ключевым словом mutable, могут быть изменены, даже если их объект объявлен как const.

На диаграмме классов UML показываются классы и отношения между ними. Объединение представляет собой концептуальное взаимоотношение между объектами реального мира, которые в программе представлены в виде классов. Объединение может иметь направление от одного класса к другому: это называется навигацией.

Вопросы

Ответы на эти вопросы вы сможете найти в приложении Ж.

1. Перегрузка операций:

   а) преобразовывает операции(operator) C++ для работы с объектами;

   б) предоставляет операциям C++ больше, чем они могут обработать;

в) дает новое значение существующим в C++ операциям;

г) создает новые операции C++.

2. Предположим, что класс X не использует перегруженные операции. Напишите выражение, в котором вычитается объект x1 класса X из другого объекта x2 этого же класса. Результат помещается в x3.
3. Предположим, что класс X включает в себя процедуру перегрузки операции –. Напишите выражение, которое будет выполнять те же действия, что и выражение, описанное в вопросе 2.
4. Истинно ли следующее утверждение: операция >= может быть перегружена?
5. Запишите полное определение для перегруженной операции в классе Counter программы COUNTPP1, которая вместо увеличения счетчика уменьшает его.
6. Сколько аргументов требуется для определения перегруженной унарной операции?
7. Предположим, что существует класс C с объектами obj1, obj2 и obj3. Выражение obj3=obj1-obj2 работает правильно. Здесь перегруженная операция должна:

   а) принимать два аргумента;

   б) возвращать значение;

   в) создавать именованный временный объект;

   г) использовать объект, вызвавший операцию, как операнд.
8. Напишите полное определение перегруженной операции ++ для класса Distance из программы ENGLPLUS. Операция должна увеличивать на единицу переменную класса feet, и при этом должно быть верным выражение:

```
dist++;
```

9. Повторите задание из вопроса 8, но так, чтобы стало возможным выполнение выражения

```
dist2 = dist1++;
```

10. Чем отличается действие перегруженной операции ++ при ее использовании в префиксной форме от использования в постфиксной форме?
11. Вот два объявления, предназначенные для складывания двух объектов класса String:

```
void add (String s1, String s2)
String operator + (String s)
```

Рассмотрим, какие из элементов первого объявления соответствуют элементам второго. Заполните пробелы подходящими вариантами:

- Имени функции (add) соответствует ________________.
- Возвращаемому значению (типа void) соответствует ______________.

- Первому аргументу (s1) соответствует ______________.
- Второму аргументу (s2) соответствует ____________.
- Объекту, вызвавшему функцию, соответствует____________.

а) аргумент (s);

б) объект, вызывающий операцию;

в) операция (+);

г) возвращаемое значение (типа String);

д) нет соответствия для этого элемента.

12. Истинно ли следующее утверждение: перегруженная операция всегда требует на один аргумент меньше, чем количество операндов?
13. Когда вы перегружаете операцию арифметического присваивания, то результат:

    а) передается объекту справа от операции;

    б) передается объекту слева от операции;

    в) передается объекту, вызвавшему операцию;

    г) должен быть возвращен.
14. Напишите полное определение перегруженной операции ++, которая работает с объектами класса String из примера STRPLUS и выполняет изменение шрифта операнда на прописные буквы. Вы можете использовать библиотечную функцию toupper() (заголовочный файл CCTYPE), принимающую в качестве аргумента символ, который надо изменить, и возвращающую уже измененный (или тот же, если изменение не нужно).
15. Для преобразования от определенного пользователем класса к основному типу вы можете использовать:

    а) встроенную операцию преобразования;

    б) конструктор с одним аргументом;

    в) перегруженную операцию =;

    г) операцию преобразования, являющуюся членом класса.
16. Истинно ли следующее утверждение: выражение objA=objB будет причиной ошибки компилятора, если объекты разных типов?
17. Для преобразования от основного типа к определенному пользователем вы можете использовать:

    а) встроенную операцию преобразования;

    б) конструктор с одним аргументом;

    в) перегруженную операцию =;

    г) операцию преобразования, являющуюся членом класса.
18. Истинно ли следующее утверждение: если вы определили конструктор, содержащий определение типа aclass obj=intvar;, вы также можете записать выражение типа obj=intvar;?

19. Если объект objA принадлежит классу A, объект objB принадлежит классу B, вы хотите записать objA=objB и поместить функцию преобразования в класс A, то какой тип процедуры преобразования вы можете использовать?
20. Истинно ли следующее утверждение: компилятор не будет протестовать, если вы перегрузите операцию * для выполнения деления?
21. В диаграммах UML объединение возникает, когда:
    а) в одной программе существуют два класса;
    б) один класс происходит от другого;
    в) в двух классах используется одна глобальная переменная;
    г) один из классов вызывает метод другого класса.
22. В UML поля классов называют ____________, а методы классов называют ______.
23. Истинно ли следующее утверждение: у прямоугольников, обозначающих классы, скругленные углы?
24. Направленность от класса A к классу B означает, что:
    а) объект класса A может вызвать операцию объекта класса B;
    б) существует взаимоотношение между классом A и классом B;
    в) объекты могут переходить из класса A в класс B;
    г) сообщения из класса B получает класс A.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Добавьте в класс Distance из программы ENGLPLUS этой главы перегруженную операцию -, которая вычисляет разность двух интервалов. Она должна позволять выполнение выражений типа dist3=dist1-dist2;. Предполагаем, что эта операция никогда не будет использоваться для вычитания большего интервала из меньшего (так как отрицательного интервала быть не может).

*2. Напишите программу, которая заменяет перегруженную операцию + на перегруженную операцию += в программе STRPLUS этой главы. Эта операция должна позволять записывать выражения типа:

```
s1 += s2;
```

где s2 прибавляется (объединяется) к строке s1, результат при этом остается в s1. Операция должна также позволять использовать результат для других вычислений, например в выражениях типа

```
s3 = s1 += s2;
```

*3. Модифицируйте класс time из упражнения 3 главы 6 так, чтобы вместо метода add_time() можно было использовать операцию + для складывания двух значений времени. Напишите программу для проверки класса.

*4. Создайте класс Int, основанный на упражнении 1 из главы 6. Перегрузите четыре целочисленных арифметических операции (+, -, * и /) так, чтобы их можно было использовать для операций с объектами класса Int. Если результат какой-либо из операций выходит за границы типа int (в 32-битной системе), имеющие значение от 2 14 7 483 648 до −2 147 483 648, то операция должна послать сообщение об ошибке и завершить программу. Такие типы данных полезны там, где ошибки могут быть вызваны арифметическим переполнением, которое недопустимо. Подсказка: для облегчения проверки переполнения выполняйте вычисления с использованием типа long double. Напишите программу для проверки этого класса.

5. Пополните класс time, рассмотренный в упражнении 3, перегруженными операциями увеличения (++) и уменьшения (--), которые работают в обеих, префиксной и постфиксной, формах записи и возвращают значение. Дополните функцию main(), чтобы протестировать эти операции.

6. Добавьте в класс time из упражнения 5 возможность вычитать значения времени, используя перегруженную операцию -, и умножать эти значения, используя тип float и перегруженную операцию *.

7. Модифицируйте класс fraction в четырехфункциональном дробном калькуляторе из упражнения 11 главы 6 так, чтобы он использовал перегруженные операции сложения, вычитания, умножения и деления. (Вспомните правила арифметики с дробями в упражнении 12 главы 3 «Циклы и ветвления».) Также перегрузите операции сравнения == и != и используйте их для выхода из цикла, когда пользователь вводит 0/1, 0 и 1 значения двух частей дроби. Вы можете модифицировать и функцию lowterms() так, чтобы она возвращала значение ее аргумента, уменьшенное до несократимой дроби. Это будет полезным в арифметических функциях, которые могут быть выполнены сразу после получения ответа.

8. Модифицируйте класс bMoney из упражнения 12 главы 7 «Массивы и строки», включив арифметические операции, выполненные с помощью перегруженных операций:

```
bMoney = bMoney + bMoney
bMoney = bMoney - bMoney
bMoney = bMoney * long double (цена за единицу времени, затраченного на изделие)
long double = bMoney / bMoney (общая цена, деленная на цену за изделие)
bMoney = bMoney / long double (общая цена, деленная на количество изделий)
```

Заметим, что операция / перегружена дважды. Компилятор может различить оба варианта, так как их аргументы разные. Помним, что легче выполнять арифметические операции с объектами класса bMoney, выполняя те же операции с его long double данными.

Убедитесь, что программа main() запросит ввод пользователем двух денежных строк и числа с плавающей точкой. Затем она выполнит все пять операций и выведет результаты. Это должно происходить в цикле, так, чтобы пользователь мог ввести еще числа, если это понадобится.

Некоторые операции с деньгами не имеют смысла: bMoney*bMoney не представляет ничего реального, так как нет такой вещи, как денежный квадрат; вы не можете прибавить bMoney к long double (что же будет, если рубли сложить с изделиями?). Чтобы сделать это невозможным, скомпилируйте такие неправильные операции, не включая операции преобразования для bMoney в long double или long double в bMoney. Если вы это сделаете и запишете затем выражение типа:

```
bmon2 = bmon1 + widgets; //это не имеет смысла
```

то компилятор будет автоматически преобразовывать widgets в bMoney и выполнять сложение. Без них компилятор будет отмечать такие преобразования как ошибки, что позволит легче найти концептуальные ошибки. Также сделайте конструкторы преобразований явными.

Вот некоторые другие вероятные операции с деньгами, которые мы еще не умеем выполнять с помощью перегруженных операций, так как они требуют объекта справа от знака операции, а не слева:

```
long double * bMoney // Пока не можем это сделать: bMoney возможен только справа
long double / bMoney // Пока не можем это сделать: bMoney возможен только справа
```

Мы рассмотрим выход из этой ситуации при изучении дружественных функций в главе 11.

9. Дополните класс safearray из программы ARROVER3 этой главы так, чтобы пользователь мог определять и верхнюю, и нижнюю границы массива (например, индексы, начинающиеся с 100 и заканчивающиеся 200). Имеем перегруженную операцию доступа к членам массива, проверяющую индексы каждый раз, когда к массиву нужен доступ, для проверки того, что мы не вышли за пределы массива. Вам понадобится конструктор с двумя аргументами, который определяет верхнюю и нижнюю границы. Так как мы еще не изучили, как выделять память динамически, то данные класса все еще будут размещаться в массиве, состоящем из 100 элементов, но вообще вы можете преобразовывать индексы массива safearray в индексы реального массива целых чисел произвольным образом. Например, если пользователь определил диапазон от 100 до 175, то вы можете преобразовать его в диапазон от arr[0] до arr[75].
10. Только для любителей математики: создайте класс Polar, который предназначен для хранения полярных координат (радиуса и угла). Перегрузите операцию + для выполнения сложения для объектов класса Polar. Сложение двух объектов выполняется путем сложения координат X объектов, а затем координат Y. Результат будет координатами новой точки. Таким образом, вам нужно будет преобразовать полярные координаты к прямоугольным, сложить их, а затем обратно преобразовать прямоугольные координаты результата к полярным.
11. Помните структуру sterling? Мы встречались с ней в упражнении 10 главы 2 «Основы программирования на C++», в упражнении 11 главы 5 и

в других местах. Преобразуйте ее в класс, имеющий переменные для фунтов (типа long), шиллингов (типа int) и пенсов (типа int). Создайте в классе следующие функции:

- конструктор без аргументов;
- конструктор с одним аргументом типа double (для преобразования от десятичных фунтов);
- конструктор с тремя аргументами: фунтами, шиллингами и пенсами;
- метод getSterling() для получения от пользователя значений количества фунтов, шиллингов и пенсов в формате £9.19.11;
- метод putSterling() для вывода значений количества фунтов, шиллингов и пенсов в формате £9.19.11;
- метод для сложения (sterling + sterling), используя перегруженную операцию +;
- метод вычитания (sterling - sterling), используя перегруженную операцию -;
- метод умножения (sterling * double), используя перегруженную операцию *;
- метод деления (sterling / sterling), используя перегруженную операцию /;
- метод деления (sterling / double), используя перегруженную операцию /;
- операцию double (для преобразования к типу double)

Выполнять вычисления вы можете, например, складывая отдельно данные объекта: сложить сначала пенсы, затем шиллинги и т. д. Однако легче использовать операцию преобразования для преобразования объекта класса sterling к типу double, выполнить вычисления с типами double, а затем преобразовать обратно к типу sterling. Таким образом, операция + выглядит похожей на эту:

```
sterling sterling::operator + (sterling s2)
{
return sterling (double (sterling (pounds, shillings, pense)) + double (s2));
}
```

Так мы создадим две временных переменных типа double, одна происходит от объекта, который вызывает функцию, а другая от аргумента s2. Эти переменные затем складываются, результат преобразовывается к типу sterling и возвращается.

Заметим, что мы использовали другой подход для класса sterling, нежели для класса bMoney. В классе sterling мы используем операции преобразования, таким образом отказавшись от возможности поиска неправильных операций, но получив простоту при записи перегружаемых математических операций.

12. Напишите программу, объединяющую в себе классы bMoney из упражнения 8 и sterling из упражнения 11. Напишите операцию преобразования для преобразования между классами bMoney и sterling, предполагая, что один фунт (£1.0.0) равен пятидесяти долларам ($50.00). Это приблизительный курс обмена для XIX века, когда Британская империя еще использовала меру фунты-шиллинги-пенсы. Напишите программу main(), которая позволит пользователю вводить суммы в каждой из валют и преобразовывать их в другую валюту с выводом результата. Минимизируйте количество изменений в существующих классах bMoney и sterling.

Глава 9
Наследование

- ✦ Базовый и производный классы
- ✦ Конструкторы производного класса
- ✦ Базовые функции класса
- ✦ Какой из методов использовать?
- ✦ Наследование в классе Distance
- ✦ Иерархия классов
- ✦ Наследование и графика
- ✦ Общее и частное наследование
- ✦ Уровни наследования
- ✦ Множественное наследование
- ✦ Частное наследование в программе EMPMULT
- ✦ Неопределенность при множественном наследовании
- ✦ Включение: классы в классах
- ✦ Роль наследования при разработке программ

Наиболее значимой после классов возможностью ООП является *наследование*. Это процесс создания новых классов, называемых *наследниками* или *производными классами*, из уже существующих или базовых классов. Производный класс получает все возможности базового класса, но может также быть усовершенствован за счет добавления собственных. Базовый класс при этом остается неизменным. Взаимосвязь классов при наследовании показана на рис. 9.1.

Возможно, что стрелка на рисунке показывает совершенно иное направление, чем вы предполагали. Если она показывает вниз, то это называется наследованием. Однако обычно она указывает вверх, от произвольного класса к базовому, и называется *производный от*.

Наследование — важная часть ООП. Выигрыш от него состоит в том, что наследование позволяет использовать существующий код несколько раз. Имея написанный и отлаженный базовый класс, мы можем его больше не модифицировать, при этом механизм наследования позволит нам приспособить его для

работы в различных ситуациях. Используя уже написанный код, мы экономим время и деньги, а также увеличиваем надежность программы. Наследование может помочь и при начальной постановке задачи программирования, разработке общей структуры программы.

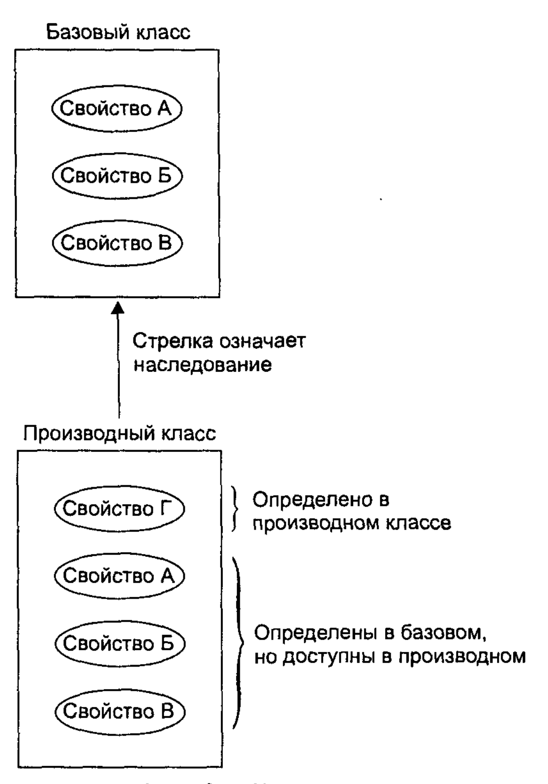


Рис. 9.1. Наследование

Важным результатом повторного использования кода является упрощение распространения библиотек классов. Программист может использовать классы, созданные кем-то другим, без модификации кода, просто создавая производные классы, подходящие для частной ситуации.

Мы рассмотрим эти возможности наследования более детально после того, как познакомимся с некоторыми тонкостями, которые могут встретиться при работе с наследованием.

Базовый и производный классы

Помните пример COUNTPP3 из главы 8 «Перегрузка операций»? Эта программа использует объект класса Counter как счетчик. Он мог быть инициализирован нулем или некоторым числом при использовании конструктора, увеличен методом operator++() и прочитан с помощью метода get_count().

Предположим, что мы затратили много времени и сил на создание класса Counter и он работает именно так, как мы хотим. Мы вполне довольны результатом, за исключением одной вещи: нам очень нужен метод для уменьшения счетчика. Возможно, мы производим подсчет посетителей банка в конкретный момент времени. При входе посетителя счетчик увеличивает свое значение, при выходе — уменьшает.

Мы могли бы вставить метод уменьшения прямо в исходный код класса Counter. Однако существует несколько причин, по которым мы не можем себе этого позволить. Во-первых, класс Counter прекрасно работает, на его отладку затрачена масса времени (конечно, в данном случае это преувеличение, но подобная ситуация может иметь место для более сложных и больших классов). Если мы вмешаемся в исходный код класса Counter, то его тестирование и отладку придется проводить вновь, тратя на это время.

Во-вторых, в некоторых ситуациях мы просто не имеем доступа к исходному коду класса, например, если он распространяется как часть библиотеки классов (мы обсудим это позднее в главе 13 «Многофайловые программы»).

Во избежание этих проблем мы можем использовать наследование для создания классов на базе Counter. Ниже представлен листинг программы COUNTEN, в которой появится новый класс CountDn, добавляющий операцию уменьшения к классу Counter[1]:

```
// counten.cpp
//inheritance with Counter class
#include <iostream.h>
using namespace std:
///////////////////////////////////////////////////////////
class Counter                  //базовый класс
{
  protected:
    unsigned int count:        //счетчик
  public:
    Counter ( ) : count ( 0 ) //конструктор без аргументов
      { }
    Counter ( int c ) : count ( c )
      { }
    unsigned int get_count ( ) const
      { return count; }        // возвращает значение счетчика
    Counter operator++ ( )     //увеличивает значение
                               //счетчика (префикс)
      { return Counter ( ++count ): }
}:
///////////////////////////////////////////////////////////
class CountDn : public Counter//производный класс
{
  public:
    Counter operator-- ( )     //уменьшает значение счетчика
      { return Counter ( --count ): }
}:
///////////////////////////////////////////////////////////
int main ( )
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
{
  CountDn c1;                              // объект c1
  cout << "\n c1=" << c1.get_count ( ); //вывод на печать
  ++c1; ++c1; ++c1;                        //увеличиваем c1 три раза
  cout << "\n c1=" << c1.get_count ( ); //вывод на печать
  --c1; --c1; //уменьшаем c1 два раза
  cout << "\n c1=" << c1.get_count ( ); //вывод на печать
  cout << endl;
  return 0;
}
```

Программа начинается с описания класса Count, которое не изменилось с момента его первого появления в COUNTPP3 (с одним небольшим исключением, которое мы рассмотрим позднее).

Заметим, что для простоты мы не включаем в эту программу операцию постфиксного увеличения, требующую использования вторичной перегрузки операции ++.

Определение производного класса

Вслед за описанием класса Count в программе определен новый класс, CountDn. Он включает в себя новый метод operator--(), который уменьшает счетчик. В то же время CountDn наследует все возможности класса Counter: конструктор и методы.

В первой строке описания класа CountDn указывается, что он является производным классом от Counter.

```
class CountDn : public Counter
```

Для этого используется знак двоеточия (не путать с двойным двоеточием, которое является операцией, использующейся для определения области действия), за ним следует ключевое слово public и имя базового класса Counter. Таким образом, мы установили отношение между классами. Другими словами, эта строка говорит нам о том, что CountDn является наследником класса Counter (значение ключевого слова public мы рассмотрим позднее).

Обобщение в диаграммах классов в UML

В UML наследование называют обобщением, так как базовый класс — это более общая форма его производных классов. Другими словами, наследник является более специализированной версией порождающего класса. (Мы познакомились с UML в главе 1 «Общие сведения» и встречались с диаграммами классов в главе 8 «Перегрузка операций».) Диаграмма обобщения для программы COUNTEN показана на рис. 9.2.

В диаграммах UML обобщение показано стрелкой с треугольным окончанием, соединяющей базовый и производный классы. Вспомним, что стрелка означает *наследование от*, *производный от* или *более специализированная версия*. Направление стрелки подчеркивает, что производный класс ссылается на методы и поля базового класса, но при этом базовый класс не имеет доступа к производному.

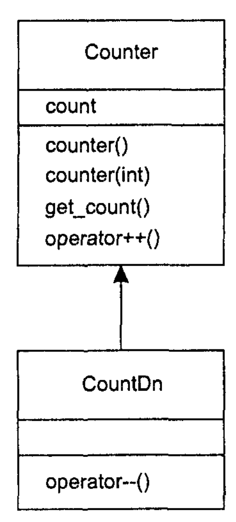


Рис. 9.2. Диаграмма классов UML для примера COUNTEN

Заметим, что на диаграмме отражены поля и методы классов. Рассмотрим элементы диаграммы, представляющие собой классы. В верхней части прямоугольника, обозначающего класс, расположено название, ниже — поля, а еще ниже — методы.

Доступ к базовому классу

Знание того, когда для объектов производного класса могут быть использованы методы базового класса, является важной темой в наследовании. Это называется *правами доступа*. Рассмотрим, как компилятор использует их в примере COUNTEN.

Подстановка конструкторов базового класса

Создадим объект класса CountDn в функции main():

```
CountDn c1;
```

Эта строка означает, что c1 будет создан как объект класса CountDn и инициализирован нулем. Но в классе CountDn нет конструктора, каким же образом выполняется инициализация? Оказывается, что если мы не определили конструктор производного класса, то будет использоваться подходящий конструктор базового класса. В COUNTEN конструктор класса CountDn отсутствует, и компилятор использует конструктор класса Counter без аргументов.

Такая гибкость компилятора — использование доступного метода взамен отсутствующего — обычная ситуация, возникающая при наследовании.

Подстановка методов базового класса

Объект c1 класса CountDn также может использовать методы operator++() и get_count() из базового класса. Сначала используем увеличение c1:

```
++c1;
```

Затем выведем на экран значение счетчика c1:

```
cout << "\n c1 = " << c1.get_count ( );
```

Вновь компилятор, не найдя этих методов в классе, объектом которого является c1, использует методы базового класса.

Результат программы COUNTEN

В функции main() происходят следующие действия: мы трижды увеличиваем c1, выводим значение на экран, затем дважды уменьшаем и вновь выводим на экран.

```
c1 = 0        - после инициализации
c1 = 3        - после увеличения c1
c1 = 1        - после уменьшения c1
```

Конструкторы, методы operator++() и get_count() класса Counter и метод operator--() класса CountDn работают с объектом класса CountDn

Спецификатор доступа protected

Мы научились расширять возможности класса почти без модификации его кода. Рассмотрим, что же мы все-таки изменили в классе Counter.

Данные классов, с которыми мы ранее познакомились, включая переменную counter класса Counter из примера COUNTPP3, имели спецификатор доступа private. В программе COUNTEN переменная counter имеет другой спецификатор доступа: protected. Рассмотрим его предназначение.

Вспомним, что нам известно о спецификаторах доступа private и public. Из методов класса есть доступ к членам (полям и методам) класса, если они имеют любой из этих спецификаторов. Но при использовании объекта, объявленного в программе, можно получить доступ только к данным со спецификатором public (например, используя операцию точки). Предположим, что есть объект objA класса A. Метод funcA() является методом класса A. Оператор функции main() (или любой другой функции, не являющейся методом класса A)

```
objA.funcA ( );
```

будет ошибочным, пока мы не объявим funcA() как public. Для объекта objA мы не можем использовать члены класса A, объявленные как private. Их могут использовать только методы самого класса A. Это показано на рис. 9.3.

Однако при использовании наследования у нас появляется еще ряд добавочных возможностей. Возникает вопрос, могут ли методы производного класса иметь доступ к членам базового класса? Другими словами, может ли operator--() класса CountDn иметь доступ к полю count класса Counter? Ответ будет таким: методы производного класса имеют доступ к членам базового класса, если они имеют спецификатор доступа public или protected. К членам, объявленным как private, доступа нет.

Мы не хотим объявлять поле count как public, так как это разрешит доступ к ней из любой функции программы, уничтожив возможность сокрытия данных. Член, объявленный как protected, доступен методам своего класса и методам любо-

го производного класса. При этом он не будет доступным из функций, не принадлежащих к этим классам, например из функции main(). Это показано на рис. 9.4.

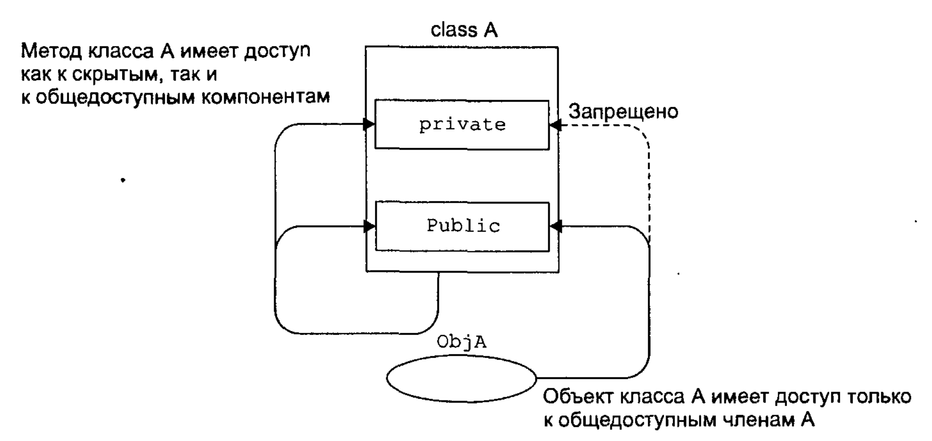


Рис. 9.3. *Спецификаторы доступа в ситуации без наследования*

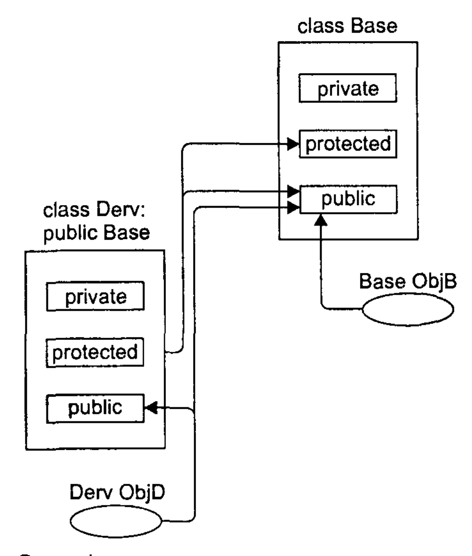


Рис. 9.4. *Спецификаторы доступа в ситуации с наследованием*

Таблица 9.1 отражает возможности использования спецификаторов доступа в различных ситуациях.

Таблица 9.1. Наследование и доступ

Спецификатор доступа	Доступ из самого класса	Доступ из производных классов	Доступ из внешних классов и функций
public	Есть	Есть	Есть
protected	Есть	Есть	Нет
private	Есть	Нет	Нет

Таким образом, если вы пишете класс, который впоследствии будет использоваться как базовый класс при наследовании, то данные, к которым нужно будет иметь доступ, следует объявлять как protected.

Недостатки использования спецификатора protected

Следует знать, что существуют и недостатки использования спецификатора доступа protected. Допустим, вы написали библиотеку классов и публично ее распространяете. Любой программист сможет получить доступ к членам классов, объявленным как protected, просто создавая производные классы. Это делает члены, объявленные как protected, значительно менее защищенными, чем объявленные как private. Чтобы избежать порчи данных, часто приходится разрешать доступ производным классам только к тем методам базового класса, которые объявлены как public. Однако использование спецификатора доступа protected упрощает программирование, и мы будем использовать эту возможность в дальнейших примерах. В будущем, при создании собственных программ, вам придется сделать выбор между преимуществами и недостатками спецификатора protected.

Неизменность базового класса

Вспомним, что при наследовании базовый класс остается неизменным. В функции main() программы COUNTEN мы определили объект типа Counter:

```
Counter c2;                  // объект базового класса
```

Такие объекты ведут себя так, как если бы класс CountDn не существовал.

Заметим также, что наследование не работает в обратном направлении. Базовому классу и его объектам недоступны производные классы. В нашем случае это означает, что объекты класса Counter, такие, как c2, не могут использовать метод operator--() класса CountDn. Если мы хотим иметь возможность уменьшения счетчика, то объект должен быть класса CountDn, а не класса Counter.

Разнообразие терминов

В некоторых языках базовый класс называют надклассом, а производный — подклассом. Некоторые ссылаются на базовый класс как на родителя, а на производный класс — как на потомка.

Конструкторы производного класса

Это потенциальная проблема в программе COUNTEN. Что будет, если мы захотим инициализировать значением объект класса CountDn? Сможем ли мы воспользоваться конструктором класса Counter с одним аргументом? Ответ будет отрицательным. Как мы видели в программе COUNTEN, компилятор будет использовать

конструктор базового класса без аргументов. Мы должны написать новый конструктор для производного класса. Это показано в программе COUNTEN2:

```
// counten2.cpp
// конструкторы в производных классах
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Counter
{
  protected:                         // заметьте, что тут не следует использовать private
    unsigned int count;              // счетчик
  public:
    Counter ( ) : count ( )          // конструктор без параметров
      { }
    Counter ( int c ) : count ( c )  // конструктор с одним параметром
      { }
    unsigned int get_count ( ) const // получение значения
      { return count; }
    Counter operator++ ( )           // оператор увеличения
      { return Counter ( ++count ); }
};
///////////////////////////////////////////////////////////
class CountDn : public Counter
{
  public:
    CountDn ( ) : Counter ( )        // конструктор без параметров
      { }
    CountDn ( int c ) : Counter ( c )// конструктор с одним параметром
      { }
    CountDn operator-- ( )           // оператор уменьшения
      { return CountDn ( --count ); }
};
///////////////////////////////////////////////////////////
int main ( )
{
  CountDn c1;                            // переменные класса CountDn
  CountDn c2 ( 100 );

  cout << "\nc1 = " << c1.get_count ( ); // выводим значения на экран
  cout << "\nc2 = " << c2.get_count ( );

  ++c1; ++c1; ++c1;                      // увеличиваем c1
  cout << "\nc1 = " << c1.get_count ( ); // показываем результат

  --c2; --c2;                            // уменьшаем c2
  cout << "c2 = " << c2.get_count ( );   // показываем результат

  CountDn c3 = --c2;                     // создаем переменную c3 на основе c2
  cout << "\nc3 = " << c3.get_count ( ); // показываем значение

  cout << endl;

  return 0;
}
```

Программа использует два новых конструктора класса CountDn. Это конструктор без аргументов:

```
CountDn ( ) : Counter ( )
  { }
```

В этом конструкторе использована новая для нас возможность: имя функции, следующее за двоеточием. Она используется конструктором класса CountDn для вызова конструктора Counter() базового класса. Когда мы запишем в функции main()

```
CountDn c1;
```

компилятор создаст объект класса CountDn и вызовет конструктор класса CountDn для его инициализации. Конструктор в свою очередь вызовет конструктор класса Counter, который выполнит нужные действия. Конструктор CountDn() может выполнять и свои операции, кроме вызова другого конструктора, но в нашем случае это не требуется, поэтому пространство между скобками пусто.

Вызов конструктора в списке инициализации может быть лишним, но это имеет смысл. Мы хотим инициализировать поле, не важно, принадлежит оно базовому или производному классу, и перед выполнением любого оператора программы сначала будут выполнены операции конструктора.

В строке

```
CountDn c2 ( 100 );
```

функции main() используется конструктор класса CountDn с одним аргументом. Этот конструктор вызывает соответствующий конструктор с одним аргументом из базового класса:

```
CountDn ( int c ) : Counter ( c )  // параметр c передается в конструктор класса Counter
  { }
```

Такая конструкция означает, что аргумент c будет передан от конструктора CountDn() в Counter(), где будет использован для инициализации объекта.

В функции main() после инициализации объектов c1 и c2 мы увеличиваем один из них и уменьшаем другой, а затем выводим результат. Конструктор с одним аргументом также используется в выражениях присваивания:

```
CountDn c3 = --c2;
```

Перегрузка функций

Мы можем определять для производного класса методы, имеющие такие же имена, как и у методов базового класса. В этом случае имеет место перегрузка функций. Такая возможность может понадобиться, если для объектов базового и производного классов в вашей программе используются одинаковые вызовы.

Рассмотрим пример, основанный на программе STAKARAY из главы 7 «Массивы и строки». Эта программа моделировала стек, простое устройство хранения данных. Она позволяла помещать числа в стек, а затем извлекать их. При попытке отправить в стек слишком много чисел программа могла зависнуть по

причине переполнения массива st [ ]. А при попытке извлечения количества чисел, большего, чем находится в стеке, результат мог быть бессмысленным, так как начинали считываться данные, расположенные в памяти за пределами массива.

Для исправления этих дефектов мы создадим новый класс Stack2, производный от Stack. Объекты класса Stack2 ведут себя так же, как объекты класса Stack, за исключением того, что мы будем предупреждать о попытке переполнить стек или извлечь число из пустого стека. Вот листинг программы STAKEN:

```
// staken.cpp
// перегрузка функций базового и производного классов
#include <iostream>
using namespace std;
#include <process.h>      // для exit ( )
///////////////////////////////////////////////////////////
class Stack
{
  protected:              // Замечание: использовать private нельзя
    enum { MAX = 3 };     // размер стека
    int st[ MAX ];        // данные, хранящиеся в стеке
    int top;              // индекс последнего элемента в стеке
  public:
    Stack ( );            // конструктор
      { top = -1; }
    void push ( int var ) // помещение числа в стек
      { st[ ++top ] = var; }
    int pop ( )           // извлечение числа из стека
      { return st[ top-- ]; }
};
///////////////////////////////////////////////////////////
class Stack2 : public Stack
{
  public:
    void push ( int var )      // помещение числа в стек
      {
        if ( top >= MAX - 1 )  // если стек полон, то ошибка
          { cout << "\nОшибка: стек полон"; exit ( 1 ); }
        Stack::push ( var );   // вызов функции push класса Stack
      }
    int pop ( )                // извлечение числа из стека
      {
        if ( top < 0 )         // если стек пуст, то ошибка
          { cout << "\nОшибка: стек пуст\n"; exit ( 1 ); }
        return Stack::pop ( ); // вызов функции pop класса Stack
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Stacks s1;

  s1.push ( 11 ); // поместим в стек несколько чисел
  s1.push ( 22 );
  s1.push ( 33 );

  cout << endl << s1.pop ( ); // заберем числа из стека
```

```
  cout << endl << s1.pop ( );
  cout << endl << s1.pop ( );
  cout << endl << s1.pop ( ); // ой, а данных-то больше нет
  cout << endl;
  return 0;
}
```

В этой программе класс Stack тот же, что и в программе STAKARAY, за исключением того, что данные класса объявлены как protected.

Какой из методов использовать?

В классе Stack2 содержатся два метода: push() и pop(). Эти методы имеют те же имена, агументы и возвращаемые значения, что и методы класса Stack. Если мы вызываем эти методы из функции main() оператором типа:

```
S1.push ( 11 );
```

то как же компилятор поймет, какой из двух методов push() вызвать? Существует правило: если один и тот же метод существует и в базовом, и в производном классе, то будет выполнен метод производного класса. (Это верно для объектов производного класса. Объектам базового класса ничего не известно о производном классе, поэтому они всегда пользуются методами базового класса.) В этом случае мы говорим, что метод производного класса *перегружает* метод базового класса. Для нашего примера, так как s1 — это объект класса Stack2, то будет выполнен метод push() класса Stack2, а не класса Stack.

Метод push() класса Stack2 проверяет, полон ли стек: если да, то он посылает сообщение об ошибке и завершает программу; если нет, то вызывается метод push() класса Stack. Таким же образом метод pop() класса Stack2 осуществляет проверку того, пуст ли стек. Если да, то выводится сообщение об ошибке и программа завершается, если нет, то вызывается метод pop() класса Stack.

В функции main() мы помещаем в стек три числа, а извлечь пытаемся четыре. Поэтому последний вызов метода pop() приводит к ошибке

```
33
22
11
Ошибка: стек пуст
```

и программа завершается.

Операция разрешения и перегрузка функций

Как же методы push() и pop() класса Stack2 получают доступ к методам push() и pop() класса Stack? Они используют операцию разрешения :: следующим образом:

```
Stack::push ( var );
```

и

```
return Stack::pop ( );
```

В этих строках определено, что будут вызваны методы push() и pop() класса Stack. Без операции разрешения компилятор подумает, что методы push() и pop() класса Stack2 вызывают сами себя, что, в нашем случае, приведет к ошибке программы. Использование операции разрешения позволяет точно определить к какому классу относится вызываемый метод.

Наследование в классе Distance

Мы рассмотрим более сложный пример использования наследования. До сих пор в примерах этой книги, использующих класс Distance, предполагалось, что интервал может иметь только положительное значение. Обычно это используется в архитектуре. Однако, если мы будем измерять уровень Тихого океана с учетом его приливов и отливов, то нам может понадобиться отрицательное значение интервала. (Если уровень прилива меньше уровня моря, то он называется отрицательным приливом; он дает сборщикам моллюсков большее пространство обнаженного пляжа.)

Давайте создадим новый производный класс от класса Distance. В этот класс мы добавим поле для наших измерений, которое содержит знак: отрицательный или положительный. А также нам нужно будет изменить методы класса, чтобы они могли работать со знаковым значением интервала. Приведем листинг программы ENGLEN:

```
// englen.cpp
// наследование в программе перевода английских мер длины
#include <iostream>
using namespace std:
enum posneg { pos. neg }:
//////////////////////////////////////////////////////////
class Distance // класс для английских мер длины
{
  protected:   // заметьте. что private использовать нельзя
    int feet:
    float inches:
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ). inches ( 0.0 )
      { }
    // конструктор с двумя параметрами
    Distance ( int ft. float in ) : feet ( ft ). inches ( in )
      { }
    // получение значений от пользователя
    void getdist ( )
      {
        cout << "\nВведите футы: ": cin >> feet:
        cout << "Введите дюймы: ": cin >> inches:
      }
    // вывод значений на экран
    void showdist ( )
      { cout << feet << "\'" << inches << '\"': }
}:
//////////////////////////////////////////////////////////
```

```
class DistSing : public Distance // добавление знака к длине
{
  private:
    posneg sing;                 // значение может быть pos или neg
  public:
    // конструктор без параметров
    DistSign ( ) : Distance ( )
      { sign = pos; }
    // конструктор с двумя или тремя параметрами
    DistSign ( int ft, float in, posneg sg = pos ) :
      Distance ( ft, in )      // вызов конструктора базового класса
      { sign = sg; }           // начальная установка знака
    void getdist ( )           // ввод пользователем длины
      {
        Distance::getdist ( ); // вызов функции getdist из базового класса
        char ch;               // запрос знака
        cout << "Введите знак (+ или -): "; cin >> ch;
        sign = ( ch == '+' ) ? pos : neg;
      }
    void showdist ( ) const    // вывод расстояния
      {
        // вывод всей информации, включая знак
        cout << ( ( sign == pos ) ? "(+)" : "(-)" );
        Distance::showdist ( );
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  DistSign alpha;                 // используем конструктор по умолчанию
  alpha.getdist ( );              // получаем данные от пользователя

  DistSign beta ( 11, 6.25 );     // конструктор с двумя аргументами

  DistSign gamma ( 100, 5.5, neg ); // конструктор с тремя аргументами

  // выводим данные для всех переменных
  cout << "\nA = "; alpha.showdist ( );
  cout << "\nB = "; beta.showdist ( );
  cout << "\nC = "; gamma.showdist ( );
  cout << endl;
  return 0;
}
```

Здесь мы видим класс DistSign, в который добавлена возможность работы со знаковыми числами. Класс Distance в этой программе тот же, что и в предыдущих, за исключением того, что его данные объявлены как protected. Фактически, в этом случае спецификатор мог быть private, так как нет методов производного класса, нуждающихся в доступе к данным класса Distance. Но если возникнет необходимость в доступе, то его всегда можно будет осуществить.

Применение программы ENGLEN

В функции main() программы объявлены три интервала. Интервал alpha получает свое значение от пользователя, beta инициализируется как (+)11'-6.25", gamma

как (–)100'-5.5". Мы заключаем знак в круглые скобки для избежания путаницы с дефисом, отделяющим футы и дюймы. Пример вывода:

```
Enter feet: 6
Enter inches: 2.5
Enter sign ( + or - ):
alpha = (-)6' - 2.5"
beta = (+)11' - 6.25"
gamma = (-)100' - 5.5"
```

Класс DistSign является производным класса Distance. В него добавлено поле sign типа posneg. Поле sign предназначено для хранения знака интервала. Тип posneg определен в операторе enum и имеет два возможных значения — pos и neg.

Конструкторы класса DistSign

Класс DistSign имеет два конструктора, таких же, как и класс Distance. Первый не имеет аргументов, у второго либо два, либо три аргумента. Третий, необязательный, аргумент второго конструктора — это переменная sign, имеющая значение pos или neg. Значением по умолчанию является pos. Эти конструкторы позволяют нам определить объекты типа DistSign разными способами.

Оба конструктора в классе DistSign вызывают соответствующие конструкторы из класса Distance для установки значений футов и дюймов. Они также устанавливают значение поля sign. Конструктор без аргументов всегда устанавливает значение поля sign равным pos. Второй конструктор устанавливает значение поля sign как pos, если оно не определено.

Аргументы ft и in, передающиеся из функции main() второму конструктору класса DistSign, просто передаются конструктору класса Distance.

Методы класса DistSign

Добавление поля sign в класс Distance имеет значение для обоих его методов. Метод getdist() класса DistSign должен послать запрос пользователю о знаках значений футов и дюймов. Метод showdist() должен вывести знаки для футов и для дюймов. Эти методы вызывают соответствующие методы из класса Distance в строках

```
Distance::getdist ( );
```

и

```
Distance::showdist ( ):
```

Вызовы получают и выводят значения футов и дюймов. Затем методы getdist() и showdist() класса DistSign продолжают работу с полем sign.

В поддержку наследования

С++ разработан для того, чтобы создание производных классов было эффективным. Мы легко можем использовать лишь части базового класса: данные, конструкторы или методы, затем добавить нужные нам возможности и создать новый,

улучшенный класс. Заметим, что в программе ENGLEN мы не применяли дубликаты кода, а вместо этого использовали подходящие методы базового класса.

Иерархия классов

До сих пор в примерах этой главы мы использовали наследование только для добавления новых возможностей к существующим классам. Теперь рассмотрим пример, где наследование применяется для других целей, как часть первоначальной разработки программы.

В качестве примера рассмотрим базу данных служащих некоторой компании. Для упрощения ситуации в ней существует только три категории служащих: менеджеры, занятые управлением, ученые, занятые исследованиями и разработкой товара компании, и рабочие, занятые изготовлением товара.

В базе данных хранятся имена служащих всех категорий и их идентификационные номера. Однако в информации о менеджерах содержится еще и название их должности и их взносы в гольф-клубы, а в информации об ученых — количество опубликованных статей.

Пример нашей программы начинается с описания базового класса employee. Этот класс содержит фамилии служащих и их номера. Он порождает три новых класса: manager, scientist и laborer. Классы manager и scientist содержат добавочную информацию об этих категориях служащих. Это показано на рис. 9.5.

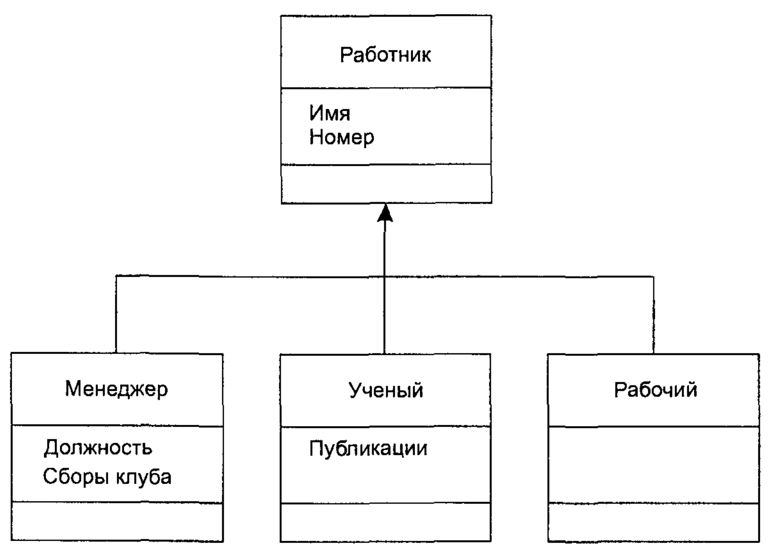


Рис. 9.5. Диаграмма классов UML для примера EMPLOY

Далее рассмотрим листинг программы EMPLOY:

```
// employ.cpp
// пример написания базы данных сотрудников с использованием наследования
#include <iostream>
```

```
using namespace std;
const int LEN = 80;        // максимальная длина имени
///////////////////////////////////////////////////////////
class employee             // некий сотрудник
{
  private:
    char name[ LEN ];      // имя сотрудника
    unsigned long number; // номер сотрудника
  public:
    void getdata ( )
      {
        cout << "\n  Введите фамилию: "; cin >> name;
        cout << "  Введите номер: ";     cin >> number;
      }
    void putdata ( ) const
      {
        cout << "\n  Фамилия: " << name;
        cout << "\n  Номер: " << number;
      }
};
///////////////////////////////////////////////////////////
class manager : public employee // менеджер
{
  private:
    char title[ LEN ];     // должность, например вице-президент
    double dues;           // сумма взносов в гольф-клуб
  public:
    void getdata ( )
      {
        employee::getdata ( );
        cout << "  Введите должность: "; cin >> title;
        cout << "  Введите сумму взносов в гольф-клуб: "; cin >> dues;
      }
    void putdata ( ) const
      {
        employee::putdata ( );
        cout << "\n  Должность: " << title;
        cout << "\n  Сумма взносов в гольф-клуб: " << dues;
      }
}
///////////////////////////////////////////////////////////
class scientist : public employee // ученый
{
  private:
    int pubs;                      // количество публикаций
  public:
    void getdata ( )
      {
        employee::getdata ( );
        cout << "  Введите количество публикаций: "; cin >> pubs;
      }
    void putdata ( ) const
      {
        employee::putdata ( );
        cout << "\n  Количество публикаций: " << pubs;
      }
};
```

```
///////////////////////////////////////////////////////////
class laborer : public employee   // рабочий
{
};
///////////////////////////////////////////////////////////
int main ( )
{
  manager m1, m2;
  scientist s1;
  laborer l1;

  // введем информацию о нескольких сотрудниках
  cout << endl;
  cout << "\nВвод информации о первом менеджере";
  m1.getdata ( );

  cout << "\nВвод информации о втором менеджере";
  m2.getdata ( );

  cout << "\nВвод информации о первом ученом";
  s1.getdata ( );

  cout << "\nВвод информации о первом рабочем";
  l1.getdata ( );

  // выведем полученную информацию на экран
  cout << "\nИнформация о первом менеджере";
  m1.putdata ( );

  cout << "\nИнформация о втором менеджере";
  m2.putdata ( );

  cout << "\nИнформация о первом ученом";
  s1.putdata ( );

  cout << "\nИнформация о первом рабочем";
  l1.putdata ( );

  cout << endl;
  return 0;
}
```

В функции main() этой программы объявлены четыре объекта различных классов: два объекта класса manager, объект класса scientist и объект класса laborer. (Конечно, объектов могло быть много больше, но при этом вывод программы стал бы довольно объемным.) Они вызывают метод getdata() для получения информации о каждом из служащих и метод putdata(), обеспечивающий вывод этой информации. Рассмотрим простое взаимодействие с программой EMPLOY. Сначала пользователь вводит данные.

```
Ввод информации о первом менеджере
  Введите фамилию: Иванов
  Введите номер: 1
  Введите должность: директор
  Сумма взносов в гольф-клуб: 10000
```

```
Ввод информации о втором менеджере
  Введите фамилию: Парнашвили
  Введите номер: 7
  Введите должность: заместитель директора
  Сумма взносов в гольф-клуб: 15000
Ввод информации о первом ученом
  Введите фамилию: Подрезов
  Введите номер: 1B
  Введите количество публикаций: 54
Ввод информации о лервом рабочем
  Введите фамилию: Сидорук
  Введите номер: 1634
Затем программа выводит введенную информацию.
Информация о первом менеджере
  Фамилия: Иванов
  Номер: 1
  Должность: директор
  Сумма взносов в гольф-клуб: 10000
Информация о втором менеджере
  Фамилия: Парнашвили
  Номер: 7
  Должность: заместитель директора
  Сумма взносов в гольф-клуб: 15000
Информация о первом ученом
  Фамилия: Подрезов
  Номер: 18
  Количество публикаций: 54
Информации о первом рабочем
  Фамилия: Сидорук
  Номер: 1634
```

Более сложная программа будет использовать для размещения данных массив или другую структуру, которая приспособлена для хранения большого количества информации о служащих.

Абстрактный базовый класс

Заметим, что мы не определяли объекты класса employee. Мы использовали его как общий класс, единственной целью которого было стать базовым для производных классов.

Класс laborer выполняет те же функции, что и класс employee, так как не имеет никаких отличий от него. Может показаться, что класс laborer в данном случае лишний, но, создав его, мы подчеркнули, что все классы имеют один источник — класс emlployee. Кроме того, если мы захотим в будущем модифицировать класс laborer, то нам не потребуется делать изменения в классе employee.

Классы, использующиеся только как базовые для производных, например как employee в программе EMPLOY, иногда ошибочно называют *абстрактными классами*, подразумевая, что у этого класса нет объектов. Однако термин *абстрактный* имеет более точное определение, и мы это увидим в главе 11 «Виртуальные функции».

Конструкторы и функции

Ни в базовом, ни в производном классах нет конструкторов, поэтому компилятор, наталкиваясь на определения типа

```
manager m1, m21;
```

использует конструктор, установленный по умолчанию для класса manager, вызывающий конструктор класса employee.

Методы getdata() и putdata() класса employee принимают от пользователя имя и номер и выводят их на дисплей. Методы getdata() и putdata() классов manager и scientist используют одноименные методы класса employee и проделывают свою работу. Метод getdata() класса manager запрашивает у пользователя должность и сумму взносов в гольф-клуб, а putdata() выводит эти значения. В классе scientist эти методы оперируют значением количества публикаций.

Наследование и графика

В программе CIRCLES главы 6 «Объекты и классы» мы видели программу, в которой класс был создан для вывода на экран кругов. Конечно, кроме кругов существует еще множество других фигур: квадраты, треугольники. Слово «фигура» является обобщением, не затрагивающим определенный тип. Используем это при создании программы более сложной, но более понятной, чем программа, которая рисует различные фигуры, не используя их общих характеристик.

В частности, мы создадим класс shape как базовый класс и три производных класса: circle, rect (для прямоугольников) и tria (для треугольников). Как и в других программах, мы используем здесь функции консольной графики. Обратитесь к приложению Д «Упрощенный вариант консольной графики», приложению В «Microsoft Visual C++» или приложению Г «Borland C++ Builder», чтобы понять, как встроить графические файлы в программу в вашем компиляторе. Приведем листинг программы MULTSHAP:

```
// multshap.cpp
// геометрические фигуры
#include <msoftcon.h>
///////////////////////////////////////////////////////////
class shape // базовый класс
{
  protected:
    int xCo, yCo;      // координаты фигуры
    color fillcolor;  // цвет
    fstyle fillstyle: // стиль изображения
  public:
    // конструктор без аргументов
    shape ( ) : xCo ( 0 ), yCo ( 0 ), fillcolor ( cWHITE ), fillstyle ( SOLID_FILL )
      { }
    // конструктор с пятью аргументами
    shape ( int x, int y, color fc, fstyle fs ) : xCo ( x ), yCo ( y ), fillcolor ( fc ),
fillstyle ( fs )
      { }
    // функция установки цвета и стиля
```

```
    void draw ( ) const
      {
        set_color ( fillcolor );
        set_fill_style ( fillstyle );
      }
};
/////////////////////////////////////////////////////////////
class circle : public shape
{
  private:
    int radius; // радиус, а xCo и yCo будут координатами центра
  public:
    // конструктор без параметров
    circle ( ) : shape ( )
      { }
    // конструктор с пятью параметрами
    circle ( int x, int y, int r, color fc, fstyle fs ) : shape ( x, y, fc, fs ), radius ( r
)
      { }
    // функция рисования окружности
    void draw ( ) const
      {
        shape::draw ( );
        draw_circle ( xCo, yCo, radius );
      }
};
/////////////////////////////////////////////////////////////
class rect : public shape
{
  private:
    int width, height; // ширина и высота, а xCo и yCo будут координатами верхнего правого
угла
  public:
    // конструктор без параметров
    rect ( ) : shape ( ), height ( 0 ), width ( 0 )
      { }
    // конструктор с шестью параметрами
    rect ( int x, int y, int h, int w, color fc, fstyle fs ) : shape ( x, y, fc, fs ),
height ( h ), width ( w )
      { }
    // функция рисования прямоугольника
    void draw ( ) const
      {
        shape::draw ( );
        draw_rectangle ( xCo, yCo, xCo + width, yCo + height );
        // нарисуем диагональ
        set_color ( xWHITE );
        draw_line ( xCo, yCo, xCo + width, yCo + height );
      }
};
/////////////////////////////////////////////////////////////
class tria : public shape
{
  private:
    int height; // высота пирамиды, а xCo и yCo будут координатами вершины
  public:
    // конструктор без параметров
    tria ( ) : shape ( ), height ( 0 )
      { }
```

```
    // конструктор с пятью параметрами
    tria ( int x, int y, int h, color fc, fstyle fs ) : shape ( x, y, fc, fs ), height ( h )
      { }
    // функция рисования пирамиды
    void draw ( ) const
      {
        shape::draw ( );
        draw_pyramid( xCo, yCo, height );
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  init_graphics ( ); // инициализируем систему отображения графики

  circle cir ( 40, 12, 5, cBLUE, X_FILL );        // создаем круг
  rect rec ( 12, 7, 10, 15, cRED, SOLID_FILL ); // создаем прямоугольник
  tria tri ( 60, 7, 11, cGREEN, MEDIUM_FILL );  // создаем пирамиду

  // рисуем все наши фигуры
  cir.draw ( );
  rec.draw ( );
  tri.draw ( );

  set_cursor_pos( 1, 25 ); // переводим курсор в самый низ экрана
  return 0;
}
```

Эта программа рисует три различных фигуры: голубой круг, красный прямоугольник и зеленый треугольник. Результат работы программы MULTSHAP показан на рис. 9.6.

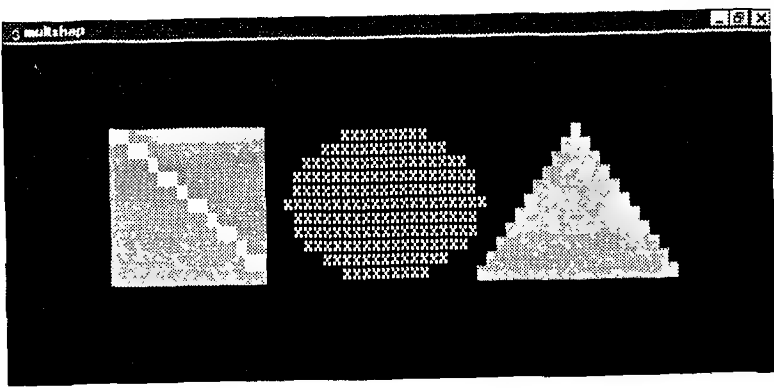


Рис. 9.6. Результат работы программы MULTSHAP

Общие характеристики всех фигур, такие, как их расположение, цвет, стиль заполнения, размещены в классе shape. Отдельные фигуры имеют более определенные атрибуты. Например, круг имеет радиус, а прямоугольник имеет ширину и высоту. Метод draw() класса shape содержит в себе действия, определенные для каждой из фигур: устанавливает их цвет и стиль заполнения. Перегрузка метода draw() в классах circle, rect и tria отвечает за прорисовку характерной для каждой фигуры формы.

Базовый класс shape является примером абстрактного класса, в котором не приписывается значение объекту класса. Вопрос, какой формы будет нарисован объект класса shape, не имеет смысла. Мы можем вывести на экран только определенную форму. Класс shape существует только как хранилище атрибутов и действий, общих для всех фигур.

Общее и частное наследование

С++ предоставляет огромное количество способов для точного регулирования доступа к членам класса. Одним из таких способов является объявление производного класса. В наших примерах мы использовали объявление типа:

```
class manager : public employee
```

которое представлено в примере EMPLOY.

Что же дает слово public в этом утверждении и имеет ли оно альтернативу? Ключевое слово public определяет, что объект производного класса может иметь доступ к методам базового класса, объявленным как public. Альтернативой является ключевое слово private. При его использовании для объектов производного класса нет доступа к методам базового класса, объявленным как public. Поскольку для объектов нет доступа к членам базового класса, объявленным как private или protected, то результатом будет то, что для объектов производных классов не будет доступа ни к одному из членов базового класса.

Комбинации доступа

Существует очень много возможностей для доступа, и полезно будет изучить программу примера, показывающую, какая комбинация будет работать, а какая нет. Рассмотрим листинг программы PUBPRIV:

```
// pubpriv.cpp
// испытание классов наследованных как public и private
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class A                   // базовый класс
{
  private:                // тип доступа к данным совпадает с типом
    int privdataA;        // доступа к функциям
  protected:
    int protdataA;
  public:
    int pubdataA;
};
///////////////////////////////////////////////////////////
class B : public A        // public наследование
{
  public:
    void funct ( )
      {
```

```
        int a;
        a = privdataA; // ошибка, нет доступа
        a = protdataA; // так можно
        a = pubdataA;  // так можно
      }
};
///////////////////////////////////////////////////////////
class C : private A    // private наследование
{
  public:
    void funct ( )
      {
        int a;
        a = privdataA; // ошибка, нет доступа
        a = protdataA; // так можно
        a = pubdataA;  // так можно
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  int a;
  B objB;
  a = objB.privdataA;  // ошибка, нет доступа
  a = objB.protdataA;  // ошибка, нет доступа
  a = objB.pubdataA;   // так можно

  C objC;
  a = objC.privdataA;  // ошибка, нет доступа
  a = objC.protdataA;  // ошибка, нет доступа
  a = objC.pubdataA;   // ошибка, нет доступа

  return 0;
}
```

В программе описан класс A, имеющий данные со спецификаторами доступа private, public и protected. Классы B и C являются производными классами. Класс B является общим наследником, а класс C — частным наследником от класса A.

Как мы видели ранее, методы производных классов имеют доступ к данным базового класса, объявленным как public или protected. Для объектов производных классов нет доступа к членам базового класса, объявленным как protected или private.

Что нового в различиях между общими и частными производным и классами? Объекты общего наследника класса B имеют доступ к членам класса A, объявленным как public или protected, а объекты частного наследника класса C имеют доступ только к членам, объявленным как public. Это показано на рис. 9.7.

Если не указывать спецификатор доступа при создании класса, то будет использован спецификатор private.

Выбор спецификатора доступа

Как же решить вопрос о том, какой из спецификаторов использовать при наследовании? В большинстве случаев производный класс представляет собой улучшенную или более специализированную версию базового класса. Мы рас-

смотрели примеры таких производных классов (вспомните класс CountDn, добавляющий новую операцию к классу Counter, или класс manager, являющийся специализированной версией класса employee). В случае, когда объект производного класса предоставляет доступ как к общим методам базового класса, так и к более специализированным методам своего класса, имеет смысл воспользоваться общим наследованием.

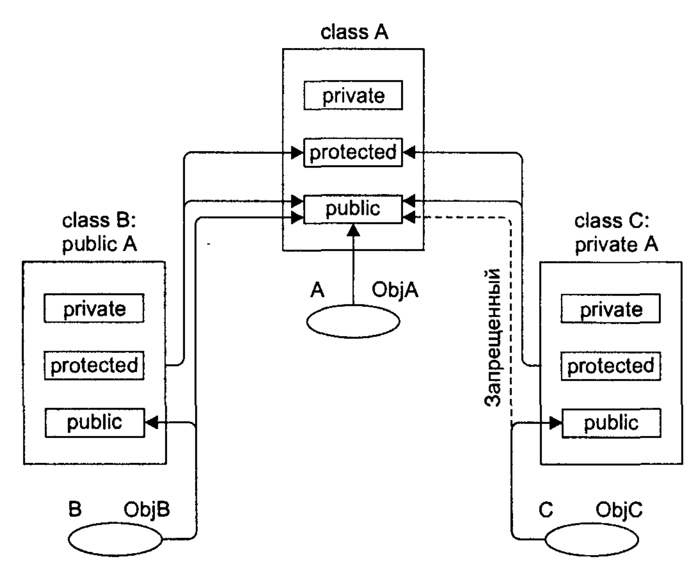


Рис. 9.7. Частное и общее наследование

Однако в некоторых ситуациях производный класс создается для полной модификации действий базового класса, скрывая или изменяя первоначальный его вид. Например, представим, что мы создали отличный класс Array, представляющий собой массив, который обеспечивает защиту против обращения к элементам, лежащим за пределами массива. Затем предположим, что мы хотим использовать класс Array как базовый для класса Stack вместо использования обычного массива. Создавая производный класс Stack, мы не хотим, чтобы с его объектами можно было работать как с массивами, например использовать операцию [ ] для доступа к элементам данных. Работа с объектами класса Stack должна быть организована как со стеками, с использованием методов push() и pop(). Тогда мы маскируем класс Array под класс Stack. В этой ситуации частное наследование позволит нам скрыть все методы класса Array от доступа через объекты производного класса Stack.

Уровни наследования

Производные классы могут являться базовыми классами для других производных классов. Рассмотрим маленькую программу в качестве примера такого случая.

```
class A
{ };
class B : public A
{ };
class C : public B
{ };
```

Здесь класс B является производным класса A, а класс C является производным класса B. Процесс может продолжаться бесконечно — класс D может быть производным класса C и т. д.

Рассмотрим более конкретный пример. Предположим, что мы решили добавить бригадиров в программу EMPLOY. Мы создадим новую программу, включающую в себя объекты класса foreman.

Класс foreman будет производным класса laborer. Это показано на рис. 9.8.

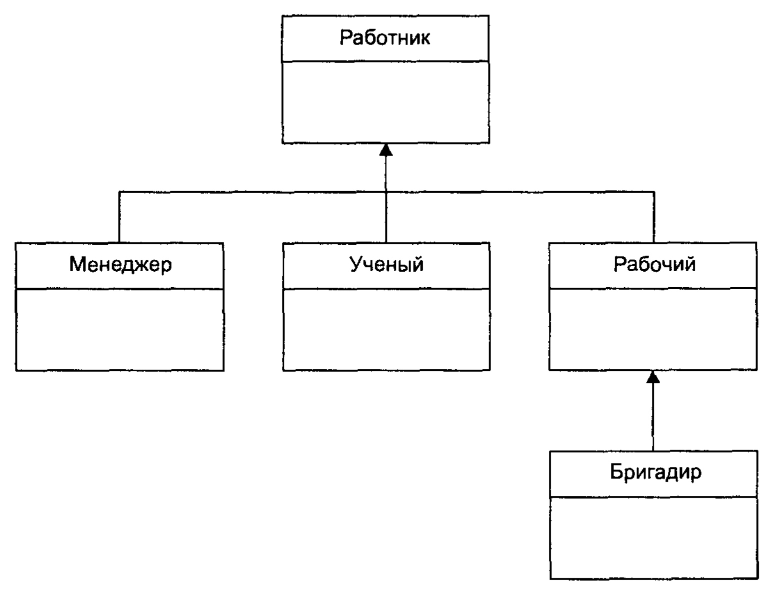


Рис. 9.8. UML диаграмма классов программы EMPLOY2

Бригадир наблюдает за выполнением работ, контролируя группу рабочих, и отвечает за выполнение нормы выработки своей группой. Эффективность работы бригадира измеряется в процентах выполнения нормы выработки. Поле quotas класса foreman представляет собой этот процент. Далее рассмотрим листинг программы EMPLOY2:

```
// employ2.cpp
// несколько уровней наследования
#include <iostream>
using namespace std;
const int LEN = 80;
////////////////////////////////////////////////////////////
class employee            // некий сотрудник
{
  private:
```

```
    char name[ LEN ]:     // имя сотрудника
    unsigned long number: // номер сотрудника
  public:
    void getdata ( )
      {
        cout << "\n  Введите фамилию: ": cin >> name:
        cout << "  Введите номер: ":     cin >> number:
      }
    void putdata ( ) const
      {
        cout << "\n  Фамилия: " << name:
        cout << "\n  Номер: " << number:
      }
}:
///////////////////////////////////////////////////////////
class manager : public employee // менеджер
{
  private:
    char title[ LEN ]:          // должность, например вице-президент
    double dues:                // сумма взносов в гольф-клуб
  public:
    void getdata ( )
      {
        employee::getdata ( ):
        cout << "  Введите должность: ": cin >> title:
        cout << "  Введите сумму взносов в гольф-клуб: ": cin >> dues:
      }
    void putdata ( ) const
      {
        employee::putdata ( ):
        cout << "\n  Должность: " << title:
        cout << "\n  Сумма взносов в гольф-клуб: " << dues:
      }
}
///////////////////////////////////////////////////////////
class scientist : public employee // ученый
{
  private:
    int pubs:                   // количество публикаций
  public:
    void getdata ( )
      {
        employee::getdata ( ):
        cout << "  Введите количество публикаций: ": cin >> pubs:
      }
    void putdata ( ) const
      {
        employee::putdata ( ):
        cout << "\n  Количество публикаций: " << pubs:
      }
}:
///////////////////////////////////////////////////////////
class laborer : public employee // рабочий
{
}:
///////////////////////////////////////////////////////////
class foreman : public laborer // бригадир
```

```
{
  private:
    float quotas;             // норма выработки
  public:
    void getdata ( )
      {
        laborer::getdata ( );
        cout << "  Введите норму выработки: "; cin >> quotas;
      }
    void putdata ( ) const
      {
        laborer::putdata ( );
        cout << "\n  Норматив: " << qoutas;
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  laborer l1;
  foreman f1;

  // введем информацию о нескольких сотрудниках
  cout << endl;
  cout << "\nВвод информации о первом рабочем";
  l1.getdata ( );
  cout << "\nВвод информации о первом бригадире";
  f1.getdata ( );

  // выведем полученную информацию на экран
  cout << endl;
  cout << "\nИнформация о первом рабочем";
  l1.putdata ( );
  cout << "\nИнформация о первом бригадире";
  f1.putdata ( );

  cout << endl;
  return 0;
}
```

Заметим, что иерархия классов отлична от схемы организации. На схеме организации показаны линии команд. Результатом иерархии классов является обобщение схожих характеристик. Чем более общим является класс, тем выше он находится в схеме. Таким образом, рабочий — это более общая характеристика, чем бригадир, который имеет определенные обязанности. Поэтому класс laborer показан над классом foreman в иерархии классов, хотя бригадир, скорее всего, имеет большую зарплату, чем рабочий.

Множественное наследование

Класс может быть производным не только от одного базового класса, а и от многих. Этот случай называется *множественным наследованием*. На рис. 9.9 показан случай, когда класс C является производным двух классов: A и B.

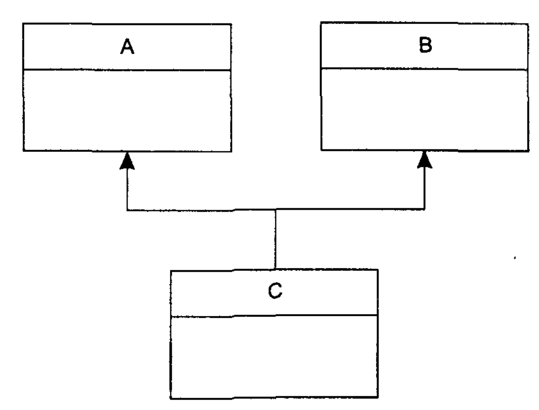


Рис. 9.9. Диаграмма классов UML при множественном наследовании

Синтаксис описания множественного наследования похож на синтаксис простого наследования. В случае, показанном на рис. 9.9, выражение будет выглядеть следующим образом:

```
class A
{
};
class B
{
};
class C : public A, public B
{
};
```

Базовые классы класса C перечислены после двоеточия в строке описания класса и разделены запятыми.

Методы классов и множественное наследование

Рассмотрим пример множественного наследования. Пусть нам для некоторых служащих необходимо указать их образование в программе EMPLOY. Теперь предположим, что в другой программе у нас существует класс student, в котором указывается образование каждого студента. Тогда вместо изменения класса employee мы воспользуемся данными класса student с помощью множественного наследования.

В классе student содержатся сведения о школе или университете, которые закончил студент, и об уровне полученного им образования. Эти данные хранятся в строковом формате. Методы getedu() и putedu() позволяют нам ввести данные о студенте и просмотреть их.

Информация об образовании нужна нам не для всех служащих. Предположим, что нам не нужны записи об образовании рабочих, а необходимы только записи об ученых и менеджерах. Поэтому мы модифицируем классы manager и scientist так, что они будут являться производными классов employee и student, как показано на рис. 9.10.

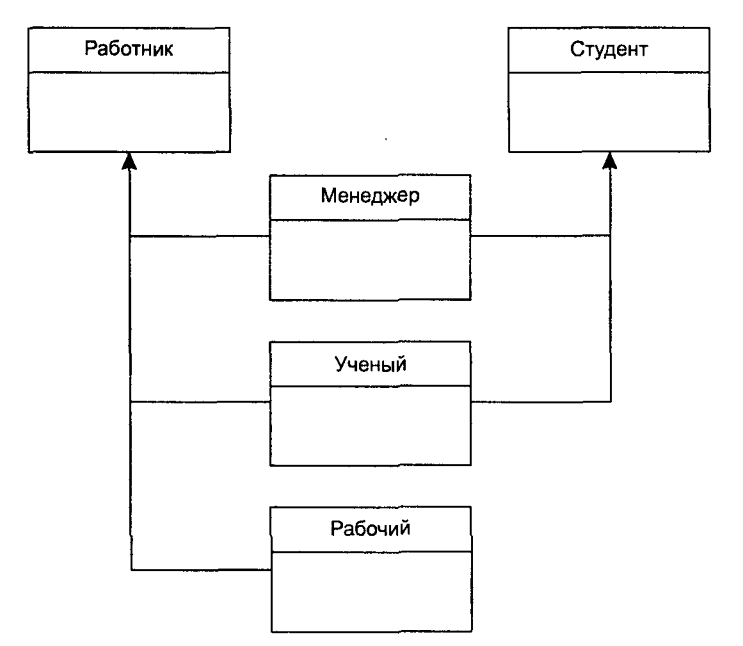


Рис. 9.10. Диаграмма классов UML программы EMPMULT

Эта маленькая программа показывает только взаимосвязь между классами:

```
class student
{ };
class employee
{ };
class manager : public employee, private student
{ };
class scientist : private employee, private student
{ };
class laborer : public employee
{ };
```

Теперь мы рассмотрим эти классы более детально в листинге EMPMULT.

```
// empmult.cpp
// множественное наследование
#incude <iostream>
usign namespace std;
const int LEN = 80;        // максимальная длина имени
////////////////////////////////////////////////////////////
class student              // сведения об образовании
{
  private:
    char school[ LEN ];   // название учебного заведения
    char degree[ LEN ];   // уровень образования
  public:
    void getedu ( )
      {
        cout << "  Введите название учебного заведения: ";
        cin >> school;
        cout << "  Введите степень высшего образования\n";
```

```
        cout << "  (неполное высшее, бакалавр, магистр, кандидат наук): ";
        cin >> degree;
      }
    void putedu ( ) const
      {
        cout << "\n  Учебное заведение: " << school;
        cout << "\n  Степень: " << degree;
      }
};
///////////////////////////////////////////////////////////
class employee            // некий сотрудник
{
  private:
    char name[ LEN ];     // имя сотрудника
    unsigned long number; // номер сотрудника
  public:
    void getdata ( )
      {
        cout << "\n  Введите фамилию: "; cin >> name;
        cout << "  Введите номер: ";     cin >> number;
      }
    void putdata ( ) const
      {
        cout << "\n  Фамилия: " << name;
        cout << "\n  Номер: " << number;
      }
};
///////////////////////////////////////////////////////////
class manager : private employee, private student // менеджер
{
  private:
    char title[ LEN ];    // должность сотрудника
    double dues;          // сумма взносов в гольф-клуб
  public:
    void getdata ( )
      {
        employee::getdata ( );
        cout << "  Введите должность: "; cin >> title;
        cout << "  Введите сумму взносов в гольф-клуб: "; cin >> dues;
        student::getedu ( );
      }
    void putdata ( ) const
      {
        employee::putdata ( );
        cout << "\n  Должность: " << title;
        cout << "\n  Сумма взносов в гольф-клуб: " << dues;
        student::putedu ( );
      }
};
///////////////////////////////////////////////////////////
class scientist : private employee, private student // ученый
{
  private:
    int pubs;             // количество публикаций
  public:
    void getdata ( )
      {
        employee::getdata ( );
        cout << "  Введите количество публикаций: "; cin >> pubs;
```

```
        student::getedu ( ):
      }
    void putdata ( ) const
      {
        employee::putdata ( ):
        cout << "\n  Количество публикаций: " << pubs:
        student::putedu ( ):
      }
}:
///////////////////////////////////////////////////////////
class laborer : public employee // рабочий
{
}:
///////////////////////////////////////////////////////////
int main ( )
{
  manager m1:
  scientist s1. s2:
  laborer l1:

  // введем информацию о нескольких сотрудниках
  cout << endl:
  cout << "\nВвод информации о первом менеджере":
  m1.getdata ( ):

  cout << "\nВвод информации о первом ученом":
  s1.getdata ( ):

  cout << "\nВвод информации о втором ученом":
  s2.getdata ( ):

  cout << "\nВвод информации о первом рабочем":
  l1.getdata ( ):

  // выведем полученную информацию на экран
  cout << "\nИнформация о первом менеджере":
  m1.putdata ( ):

  cout << "\nИнформация о первом ученом":
  s1.putdata ( ):

  cout << "\nИнформация о втором ученом":
  s2.putdata ( ):

  cout << "\nИнформация о первом рабочем":
  l1.putdata ( ):

  cout << endl:
  return 0:
}
```

Функции getdata() и putdata() классов manager и scientist включают в себя такие вызовы функций класса student, как

```
student::getedu ( ):
```

и

```
student::putedu ( ):
```

Эти методы доступны классам manager и scientist, поскольку названные классы наследуются от класса student.

Рассмотрим небольшой пример работы программы EMPMULT:

```
Введите данные для менеджера 1
  Введите фамилию: Иванов
  Введите номер: 12
  Введите должность: Вице-президент
  Введите сумму взносов в гольф-клуб: 1000000
  Ведите название школы или университета: СПбГУ
  Введите полученную ученую степень: бакалавр

Введите данные для ученого 1
  Введите фамилию: Петров
  Введите номер: 764
  Введите количество публикаций: 99
  Введите название школы или университета: МГУ
  Введите полученную ученую степень: кандидат наук

Введите данные для ученого 2
  Введите фамилию: Сидоров
  Введите номер: 845
  Введите количество публикаций: 101
  Введите название школы или университета: МГУ
  Введите полученную ученую степень: доктор наук

Введите данные для рабочего 1
  Введите фамилию: Строкин
  Введите номер: 48323
```

Как мы видим, программы EMPLOY и EMPLOY2 работают примерно одинаково.

Частное наследование в программе EMPMULT

Классы manager и scientist являются частными производными классов employee и student. В этом случае нет надобности использовать общее наследование, так как объекты классов manager и scientist не пользуются методами базовых классов employee и student. Однако класс laborer должен быть общим производным класса employee, поскольку он пользуется его методами.

Конструкторы при множественном наследовании

Программа EMPMULT не имеет конструкторов. Рассмотрим пример, показывающий, как работают конструкторы при множественном наследовании.

Представим, что мы пишем программу для строителей-подрядчиков, которая работает со строительными материалами. Нам нужен класс, определяющий количество стройматериалов каждого типа, например 100 восьмиметровых бревен. Другой класс должен хранить различные данные о каждом виде стройматериа-

лов. Например, длину куска материала, количество таких кусков и стоимость за погонный метр.

Нам также нужен класс, хранящий описание каждого вида стройматериала, которое включает в себя две части. В первой части номинальные размеры поперечного сечения материала. Они измеряются дюймами. Во второй — сорт материала. Класс включает в себя строковые поля, которые описывают номинальные размеры и сорт материала. Методы класса получают информацию от пользователя, а затем выводят ее на экран.

Мы используем класс Distance из предыдущего примера для хранения длины материала. Наконец, мы создали класс Lumber который является производным классов Type и Distance. Далее рассмотрим листинг программы ENGLMULT.

```
// englmult.cpp
// программа демонстации множественного наследования
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class Type                  // Тип древесины
{
  private:
    string dimensions;
    string grade;
  public:
    // конструктор без параметров
    Type ( ) : dimensions ( "N/A" ), grade ( "N/A" )
      { }
    // конструктор с двумя параметрами
    Type ( string di, string gr ) : dimensions ( di ), grade ( gr )
      { }
    void gettype ( )        // запрос информации у пользователя
      {
        cout << "  Введите номинальные размеры (2x4 и т.д.): ";
        cin >> dimensions;
        cout << "  Введите сорт (необработанная, брус и т.д.): ";
        cin >> grade;
      }
    void showtype ( )       // показ информации
      {
        cout << "\n  Размеры: " << dimensions;
        cout << "\n  Сорт: " << grade;
      }
};
///////////////////////////////////////////////////////////
class Distance              // английские меры длины
{
  private:
    int feet;
    float inches;
  public:
    // конструктор без параметров
    Distance ( ) : feet ( 0 ), inches ( 0.0 )
      { }
    // конструктор с двумя параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in )
```

```
      { }
    void getdist ( )        // запрос информации у пользоватепя
      {
        cout << "  Введите футы: "; cin >> feet;
        cout << "  Введите дюймы: "; cin >> inches;
      }
    void showdist ( ) const // показ информации
      {
        cout << feet << "\'-" << inches << '\"';
      }
};
///////////////////////////////////////////////////////////
class Lumber: public Type, public Distance
{
  private:
    int quantity;           // количество
    double price;           // цена за штуку
  public:
    // конструктор без параметров
    Lumber ( ) : Type ( ), Distance ( ), quantity ( 0 ), price ( 0.0 )
      { }
    // конструктор с шестью параметрами
    Lumber ( string di, string gr, // параметры для Type
             int ft, float in,     // параметры для Distance
             int qu, float prc ) : // наши собственные параметры
             Type ( di, gr ),      // вызов конструктора Type
             Distance ( ft, in ),  // вызов конструктора Distance
             quantity ( qu ), price ( prc ) // вызов наших конструкторов
      { }
    void getlumber ( )
      {
        Type::gettype ( );
        Distance::getdist ( );
        cout << "  Введите количество: "; cin >> quantity;
        cout << "  Введите цену: "; cin >> price;
      }
    void showlumber ( ) const
      {
        Type::showtype ( );
        cout << "\n  Длина: ";
        Distance::showdist ( );
        cout << "\n  Стоимость " << quantity << " штук: " << ( price * quantity ) << "
рублей";
      }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Lumber siding;            // используем конструктор без параметров

  cout << "\n  Информация об обшивке:\n";
  siding.getlumber ( );     // получаем данные от пользователя

  // используем конструктор с шестью параметрами
  Lumber studs ( "2x4", "const", 8, 0.0, 200, 4.45F );

  // показываем информацию
  cout << "\nОбшивка"; siding.showlumber ( );
```

```
  cout << "\nБрус";    studs.showlumber ( );

  cout << endl;
  return 0;
}
```

Главной особенностью этой программы является использование конструкторов производного класса Lumber. Они вызывают подходящие конструкторы классов Type и Distance.

Конструкторы без аргументов

В классе Type конструктор без аргументов выглядит следующим образом:

```
Type ( )
  { strcpy( dimensions, "N/A" ); strcpy( grade, "N/A" ) }
```

Этот конструктор присваивает значениям полей dimensions и grade строку "N/A" (недоступно), поэтому при попытке вывести данные для объекта класса Lumber пользователь будет знать, что поля пусты.

Нам уже знаком конструктор без аргументов в классе Distance:

```
Distance ( ) : feet ( 0 ), inches ( 0.0 )
  { }
```

Конструктор без аргументов класса Lumber вызывает конструкторы обоих классов — Type и Distance.

```
Lumber : Type ( ), Distance ( ), quantity ( 0 ), price ( 0.0 )
  { }
```

Имена конструкторов базового класса указаны после двоеточия и разделены запятыми. При вызове конструктора класса Lumber начинают работу конструкторы базовых классов Type() и Distance(). При этом инициализируются переменные quantity и price.

Конструктор со многими аргументами

Конструктор класса Type с двумя аргументами выглядит следующим образом:

```
Type ( string di, string gr ) : dimensions ( di ), grade ( gr )
  { }
```

Этот конструктор копирует строковые аргументы в поля класса dimensions и grade.

Конструктор класса Distance тот же, что и в предыдущей программе:

```
Distance ( int ft, float in ) : feet ( ft ), inches ( in )
  { }
```

В конструктор класса Lumber включены оба этих конструктора, которые получают значения для аргументов. Кроме того, класс Lumber имеет и свои аргументы: количество материала и его цена. Таким образом, конструктор имеет шесть аргументов. Он вызывает два конструктора, которые имеют по два

аргумента, а затем инициализирует два собственных поля. Конструктор класса Lumber будет выглядеть следующим образом:

```
Lumber ( string di. string gr,         // параметры для Type
         int ft. float in,             // параметры для Distance
         int qu. float prc ) :         // наши собственные параметры
         Type ( di, gr ),              // вызов конструктора Type
         Distance ( ft, in ),          // вызов конструктора Distance
         quantity ( qu ). price ( prc ) // вызов наших конструкторов
  { }
```

Неопределенность при множественном наследовании

В определенных ситуациях могут появиться некоторые проблемы, связанные со множественным наследованием. Здесь мы рассмотрим наиболее общую. Допустим, что в обоих базовых классах существуют методы с одинаковыми именами, а в производном классе метода с таким именем нет. Как в этом случае объект производного класса определит, какой из методов базовых классов выбрать? Одного имени метода недостаточно, поскольку компилятор не сможет вычислить, какой из двух методов имеется в виду.

Эту ситуацию мы разберем в примере AMBIGU:

```
// ambigu.cpp
// демонстрация неопределенности при множественном наследовании
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class A
{
  public:
    void show ( ) { cout << "Класс A\n"; }
};
class B
{
  public:
    void show ( ) { cout << "Класс B\n"; }
};
class C : public A, public B
{
};
///////////////////////////////////////////////////////////
int main ( )
{
  C objC;            // обьект класса C
  // objC.show ( ); // так делать нельзя - программа не скомпилируется
  objC.A::show ( ); // так можно
  objC.A::show ( ); // так можно
  return 0;
}
```

Проблема решается путем использования оператора разрешения, определяющего класс, в котором находится метод. Таким образом,

```
ObjC.A::show ( ):
```

направляет нас к версии метода show(), принадлежащей классу A, а

```
ObjC.B::show ( ):
```

направляет нас к методу, принадлежащему классу B. Б. Страуструп (см. приложение З «Библиография») называет это *устранением неоднозначности.*

Другой вид неопределенности появляется, если мы создаем производный класс от двух базовых классов, которые, в свою очередь, являются производными одного класса. Это создает дерево наследования в форме ромба. В программе DIAMOND показано, как это выглядит.

```
// diamond.cpp
// демонстрация наследования в форме ромба
#include <iostream>
using namespace std:
///////////////////////////////////////////////////////////
class A
{
  public:
    void func ( ):
}:
class B : public A
{ }:
class C : public A
{ }
class D : public B. public C:
///////////////////////////////////////////////////////////
int main ( )
{
  D objD:
  ObjD.func ( ): // неоднозначность: программа не скомпилируется
  return 0:
}
```

Классы B и C являются производными класса A, а класс D является производным классов B и C. Трудности начинаются, когда объект класса D пытается воспользоваться методом класса A. В этом примере объект objD использует метод func(). Однако классы B и C содержат в себе копии метода func(), унаследованные от класса A. Компилятор не может решить, какой из методов использовать, и сообщает об ошибке.

Существует множество вариаций этой проблемы, поэтому многие эксперты советуют избегать множественного наследования. Не следует использовать его в своих важных программах, пока у вас нет достаточного опыта.

Включение: классы в классах

Мы обсудим здесь включение, потому что, хотя оно и не имеет прямого отношения к наследованию, механизм включения и наследование являются формами взаимоотношений между классами. Поэтому будет полезно сравнить их.

Если класс B является производным класса A, то мы говорим, что *класс B является отчасти классом A*, так как класс B имеет все характеристики класса A и, кроме того, свои собственные. Точно также мы можем сказать, что скворец это птица, так как он имеет признаки, характерные для птиц (крылья, перья и т. д.), но при этом имеет свои собственные отличительные признаки (такие, как темное переливающееся оперение). Поэтому часто наследование называют взаимоотношением.

Включение называют взаимоотношением типа «имеет». Мы говорим, что библиотека имеет книги (в библиотеке есть книги) или накладная имеет строки (в накладной есть строки). Включение также называют взаимоотношением типа «часть целого»: книга является частью библиотеки.

В ООП включение появляется, когда один объект является атрибутом другого объекта. Рассмотрим случай, когда объект класса A является атрибутом класса B:

```
class A
{ };
class B
{
  A objA;
};
```

В диаграммах UML включение считается специальным видом объединения. Иногда трудно сказать, где включение, а где объединение. Всегда безопасно называть взаимоотношения объединением, но если класс A содержит объект класса B и превосходит класс B по организации, то это будет уже включением. Компания может иметь включение служащих или коллекция марок может иметь включение марок.

Включение в диаграммах UML показывается так же, как объединение, за исключением того, что включение имеет стрелку в виде ромба. На рис. 9.11 показано, как это выглядит.

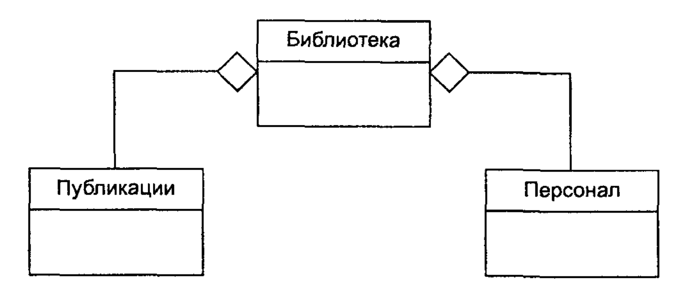


Рис. 9.11. Включение на диаграммах классов UML

Включение в программе EMPCONT

Давайте в программе EMPMULT применим включение вместо наследования. Классы manager и scientist программы EMPMULT являются производными классов employee и student. Здесь использовано наследование. В нашей новой программе

EPMCONT классы manager и scientist содержат копии классов employee и student как атрибуты. Взаимосвязи этого объединения показаны на рис. 9.12.

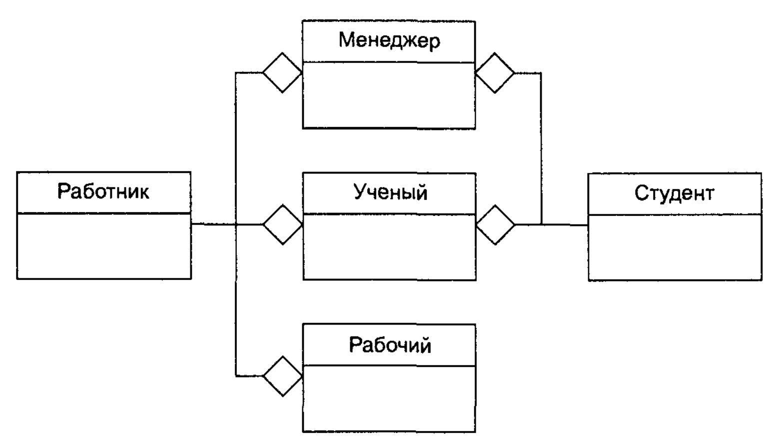


Рис. 9.12. Диаграмма классов UML для программы EMPCONT

Рассмотренная ниже небольшая программа показывает, как осуществлены эти взаимосвязи другим способом:

```
class student
{ };
class employee
{ };
class manager
{
  student stu;
  employee emp;
};
class scientist
{
  student stu;
  employee emp;
};
class laborer
{
  employee emp;
};
```

Вот полный листинг программы EMPCONT:

```
// empcont.cpp
//
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class student
{
  private:
    string school;
    string degree;
```

```
  public:
    void getedu ( )
      {
        cout << "  Введите название учебного заведения: ";
        cin >> school;
        cout << "  Введите уровень образования\n";
        cout << "  (неполное высшее, бакалавр, магистр, кандидат наук): ";
        cin >> degree;
      }
    void putedu ( ) const
      {
        cout << "\n  Учебное заведение: " << school;
        cout << "\n  Степень: " << degree;
      }
};
///////////////////////////////////////////////////////////
class employee
{
  private:
    string name;
    unsigned long number;
  public:
    void getdata ( )
      {
        cout << "\n  Введите фамилию: "; cin >> name;
        cout << "  Введите номер: ";      cin >> number;
      }
    void putdata ( ) const
      {
        cout << "\n  Фамилия: " << name;
        cout << "\n  Номер: " << number;
      }
};
///////////////////////////////////////////////////////////
class manager
{
  private:
    string title;
    double dues;
    employee emp;
    student stu;
  public:
    void getdata ( )
      {
        emp.getdata ( );
        cout << "  Введите должность: "; cin >> title;
        cout << "  Введите сумму взносов в гольф-клуб: "; cin >> dues;
        stu.getedu ( );
      }
    void putdata ( ) const
      {
        emp.putdata ( );
        cout << "\n  Должность: " << title;
        cout << "\n  Сумма взносов в гольф-клуб: " << dues;
        stu.putedu ( );
      }
};
///////////////////////////////////////////////////////////
```

```
class scientist
{
  private:
    int pubs;
    employee emp;
    student stu;
  public:
    void getdata ( )
      {
        emp.getdata ( );
        cout << "  Введите количество публикаций: "; cin >>
        stu.getedu ( );
      }
    void putdata ( ) const
      {
        emp.putdata ( );
        cout << "\n  Количество публикаций: " << pubs;
        stu.putedu ( );
      }
};
/////////////////////////////////////////////////////////////
class laborer
{
  private:
    employee emp;
  public:
    void getdata ( )
      { emp.getdata ( ); }
    void putdata ( ) const
      { emp.putdata ( ); }
};
/////////////////////////////////////////////////////////////
int main ( )
{
  manager m1;
  scientist s1, s2;
  laborer l1;

  // введем информацию о нескольких сотрудниках
  cout << endl;
  cout << "\nВвод информации о первом менеджере";
  m1.getdata ( );

  cout << "\nВвод информации о первом ученом";
  s1.getdata ( );

  cout << "\nВвод информации о втором ученом";
  s2.getdata ( );

  cout << "\nВвод информации о первом рабочем";
  l1.getdata ( );

  // выведем полученную информацию на экран
  cout << "\nИнформация о первом менеджере";
  m1.putdata ( );

  cout << "\nИнформация о первом ученом";
  s1.putdata ( );
```

```
    cout << "\nИнформация о втором ученом";
    s2.putdata ( );

    cout << "\nИнформация о первом рабочем";
    l1.putdata ( );

    cout << endl;
    return 0;
}
```

Классы employee и student в программе EMPCONT те же, что и в EMPMULT, но с классами manager и scientist они связаны другим способом.

Композиция: сложное включение

Композиция — это более сложная форма объединения. Она обладает всеми его свойствами, но имеет еще и такие, как:

- часть может принадлежать только одному целому;
- время жизни части то же, что и целого.

Машина имеет двери (помимо других деталей). Двери не могут принадлежать другой машине, они являются ее неотъемлемой частью. В комнате есть пол, потолок и стены. Если включение — это взаимоотношение типа «имеет», то композиция — это взаимоотношение типа «состоит из».

В диаграммах UML композиция показывается так же, как и включение, за исключением того, что ромб стрелки закрашен. Это представлено на рис. 9.13.

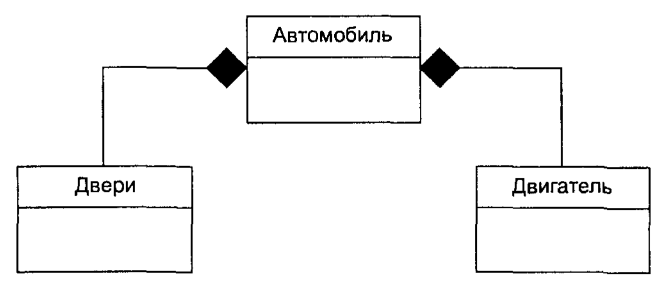


Рис. 9.13. Композиция в диаграммах классов UML

Даже одиночный объект может относиться к классу как композиция. В машине только один двигатель.

Роль наследования при разработке программ

Процесс разработки программ был основательно изменен с появлением ООП. Это связано не только с использованием классов, но и с использованием наследования. Рассмотрим его этапы.

Пусть программист А создает класс. Положим, это будет класс, похожий на класс Distance, методы которого предназначены для выполнения арифметических операций с определенным пользователем типом данных.

Программисту Б нравится этот класс, но он считает, что класс может быть улучшен путем введения знака интервала. Решением будет создание нового класса, похожего на класс DistSign программы ENGLEN, который является производным класса Distance, но включает в себя расширения, необходимые для реализации знака интервала.

Программисты В и Г затем пишут приложения, использующие класс DistSign.

Программист Б может не иметь доступа к исходному коду, реализующему класс Distance, а программисты В и Г могут не иметь исходного кода класса DistSign. Но в C++ существует возможность повторного использования кода, поэтому программист Б может использовать и дополнять работу программиста А, а В и Г могут использовать работы Б (и А).

Заметим, что различия между разработчиками инструментов программного обеспечения и программистами, пишущими приложения, становятся расплывчатыми. Программист А создает общецелевой программный инструмент, класс Distance. Программист Б, в свою очередь, создает более специализированный класс DistSign. Программисты В и Г пишут приложения. А — разработчик инструмента, а В и Г — разработчики приложений. Б находится где-то посередине. В любом случае ООП делает программирование более гибким, но в то же время и более сложным.

В главе 13 мы рассмотрим, как классы могут быть разделены на доступную клиенту часть и часть, которая распространяется только для объектов и может быть использована другими программистами без наличия исходного кода.

Резюме

Класс, называемый *производным классом*, может наследовать возможности другого класса, называемого *базовым классом*. При этом в производном классе могут быть свои собственные возможности, так как он является более специализированной версией базового класса. Наследование предоставляет эффективный способ расширения возможностей существующих классов и разработки программ с использованием иерархических связей между классами.

Важным вопросом является доступность членов базового класса для методов производных классов и объектов производных классов. К полям и методам базового класса, объявленным как protected, могут иметь доступ только методы производного класса. Объекты внешних классов, включая сам производный класс, в этом случае доступа к базовому классу не имеют. Классы могут быть общими и частными производными базового класса. Объекты общего производного класса имеют доступ к членам базового класса, объявленным как public, а объекты частного производного класса доступа к ним не имеют.

Класс может быть производным более чем одного базового класса. Этот случай называется *множественным наследованием*. Также класс может содержаться внутри другого класса.

В диаграммах UML наследование называют *обобщением*. Оно представляется в виде треугольной стрелки, указывающей на базовый класс.

Включение — это взаимоотношение типа «имеет» или «является частью», при этом один класс содержит объекты другого класса. Включение представляется на диаграммах UML стрелкой в форме ромба, показывающей на целую часть пары часть—целое. Композиция — это более сложная форма объединения. На диаграммах она показывается так же, но стрелка закрашена.

Наследование позволяет использовать код программ повторно: в производном классе можно расширить возможности базового класса без его модификации, даже не имея доступа к его коду. Это приводит к появлению гибкости в процессе разработки программного обеспечения и расширению роли программных разработчиков.

Вопросы

Ответы на эти вопросы вы сможете найти в приложении Ж.

1. Назначение наследования состоит в том, чтобы:
   а) создавать более общие классы в более специализированных;
   б) передавать аргументы объектам классов;
   в) добавлять возможности к существующим классам без их модификации;
   г) улучшать сокрытие данных и их инкапсуляцию.
2. Класс-наследник называется ____________ от базового класса.
3. Преимущество использования наследования заключается в:
   а) обеспечении развития класса путем естественного отбора;
   б) облегчении создания классов;
   в) избежании переписывания кода;
   г) предоставлении полезной концептуальной основы.
4. Напишите первую строку описания класса Bosworth, который является public-производным класса Alphonso.
5. Будет ли правильным утверждение: создание производного класса требует коренных изменений в базовом классе?
6. Члены базового класса для доступа к ним методов производного класса должны быть объявлены как public или ________.
7. Пусть базовый класс содержит метод basefunc(), а производный класс не имеет метода с таким именем. Может ли объект производного класса иметь доступ к методу basefunc()?
8. Допустим, что класс, описанный в вопросе 4, и класс Alphonso содержат метод alfunc(). Напишите выражение, позволяющее объекту BosworthObj класса Bosworth получить доступ к методу alfunc().

9. Истинно ли следующее утверждение: если конструктор производного класса не определен, то объекты этого класса будут использовать конструкторы базового класса?
10. Допустим, что базовый и производный классы включают в себя методы с одинаковыми именами. Какой из методов будет вызван объектом производного класса, если не использована операция разрешения имени?
11. Напишите объявление конструктора без аргументов для производного класса Bosworth из вопроса 4, который будет вызывать конструктор без аргументов класса Alphonso.
12. Оператор разрешения обычно:
    а) ограничивает видимость переменных для определенных методов;
    б) обозначает, от какого базового класса создан производный;
    в) определяет отдельный класс;
    г) разрешает неопределенности.
13. Истинно ли следующее утверждение: иногда полезно создать класс, объектов которого никогда не будет создано?
14. Предположим, что существует класс Derv, производный от базового класса Base. Напишите объявление конструктора производного класса, принимающего один аргумент и передающего его в конструктор базового класса.
15. Предположим, что класс Derv является частным производным класса Base. Мы определяем объект класса Derv, расположенный в функции main(). Через него мы можем получить доступ к:
    а) членам класса Derv, объявленным как public;
    б) членам класса Derv, объявленным как protected;
    в) членам класса Derv, объявленным как private;
    г) членам класса Base, объявленным как public;
    д) членам класса Base, объявленным как protected;
    е) членам класса Base, объявленным как private.
16. Истинно ли следующее утверждение: класс D может быть производным класса C, который в свою очередь является производным класса B, который производный класса A?
17. Иерархия классов:
    а) показывает те же взаимоотношения, что и схема организации;
    б) описывает взаимоотношение типа «имеет»;
    в) описывает взаимоотношения типа «является частью»;
    г) показывает те же взаимоотношения, что и наследственное дерево.
18. Напишите первую строку описания класса Tire, который является производным классов Wheel и Rubber.
19. Предположим, что класс Derv является производным класса Base. Оба класса содержат метод func() без аргументов. Напишите выражение, входящее в метод класса Derv, которое вызывает метод func() базового класса.

20. Истинно ли следующее утверждение: невозможно сделать объект одного класса членом другого класса?
21. В UML наследование называют __________.
22. Включение — это:
    а) сложная форма реализации;
    б) сложная форма обобщения;
    в) сложная форма композиции;
    г) взаимоотношение типа «имеет».
23. Истинно ли следующее утверждение: стрелка, представляющая собой обобщение, указывает на более специфичный класс?
24. Композиция — это ________форма ________.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Представьте себе издательскую компанию, которая торгует книгами и аудио-записями этих книг. Создайте класс publication, в котором хранятся название (строка) и цена (типа float) книги. От этого класса наследуются еще два класса: book, который содержит информацию о количестве страниц в книге (типа int), и type, который содержит время записи книги в минутах (тип float). В каждом из этих трех классов должен быть метод getdata(), через который можно получать данные от пользователя с клавиатуры, и putdata(), предназначенный для вывода этих данных.

   Напишите функцию main() программы для проверки классов book и type. Создайте их объекты в программе и запросите пользователя ввести и вывести данные с использованием методов getdata() и putdata().

*2. Вспомните пример STRCONV из главы 8. Класс String в этом примере имеет дефект: у него нет защиты на тот случай, если его объекты будут инициализированы слишком длинной строкой (константа SZ имеет значение 80). Например, определение

```
String s = "Эта строка имеет очень большую длину и мы можем быть уверены, что она не
уместится в отведенный буфер, что приведет к непредсказуемым последствиям.";
```

   будет причиной переполнения массива str строкой s с непредсказуемыми последствиями вплоть до краха системы.

   Создадим класс Pstring, производный от класса String, в котором предотвратим возможность переполнения буфера при определении слишком длинной строковой константы. Новый конструктор производного класса будет копировать в str только SZ-1 символов, если строка окажется слишком длинной, и будет копировать строку полностью, если она будет иметь длину меньшую, чем SZ. Напишите функцию main() программы для проверки ее работы со строками разной длины.

*3. Начните с классов book, type и publication из упражнения 1. Добавьте базовый класс sales, в котором содержится массив, состоящий из трех значений типа float, куда можно записать общую стоимость проданных книг за последние три месяца. Включите в класс методы getdata() для получения значений стоимости от пользователя и putdata() для вывода этих цифр. Измените классы book и type так, чтобы они стали производными обоих классов: publications и sales. Объекты классов book и type должны вводить и выводить данные о продажах вместе с другими своими данными. Напишите функцию main() для создания объектов классов book и type, чтобы протестировать возможности ввода/вывода данных.

4. Предположим, что издатель из упражнений 1 и 3 решил добавить к своей продукции версии книг на компьютерных дисках для тех, кто любит читать книги на своих компьютерах. Добавьте класс disk, который, как book и type, является производным класса publication. Класс disk должен включать в себя те же функции, что и в других классах. Полем только этого класса будет тип диска: CD или DVD. Для хранения этих данных вы можете ввести тип enum. Пользователь должен выбрать подходящий тип, набрав на клавиатуре c или d.

5. Создайте производный класс employee2 от базового класса employee из программы EMPLOY этой главы. Добавьте в новый класс поле compensation типа double и поле period типа enum для обозначения периода оплаты работы служащего: почасовая, понедельная или помесячная. Для простоты вы можете изменить классы laborer, manager и scientist так, чтобы они стали производными нового класса employee2. Однако заметим, что во многих случаях создание отдельного класса compensation и трех его производных классов manager2, scientist2 и laborer2 более соответствовало бы духу ООП. Затем можно применить множественное наследование и сделать так, чтобы эти три новых класса стали производными класса compensation и первоначальных классов manager, scientist и laborer. Таким путем можно избежать модификации исходных классов.

6. Вспомним программу ARROVER3 из главы 8. Сохраним класс safearray таким же и, используя наследование, добавим к нему возможность для пользователя определять верхнюю и нижнюю границы массива в конструкторе. Это похоже на упражнение 9 из главы 8, за исключением того, что применено наследование для создания нового класса (можно назвать его safehilo) взамен модификации исходного класса.

7. Вспомним программу COUNTEN2 из этой главы. В ней можно увеличивать и уменьшать счетчик, используя префиксные операции. Используя наследование, добавьте возможность использования постфиксных операций для случаев увеличения и уменьшения. (Описание постфиксных операций вы сможете найти в главе 8.)

8. В некоторых компьютерных языках, таких, как Visual Basic, есть операции, с помощью которых можно выделить часть строки и присвоить ее другой строке. (В стандартном классе string предложены различные под-

ходы.) Используя наследование, добавьте такую возможность в класс Pstring из упражнения 2. В новом производном классе Pstring2 разместите три новых функции: left(), mid() и right().

```
s2.left ( s1, n )   // в строку s2 помещаются n самых левых
                    // символов строки s1
s2.mid ( s1, s, n ) // в строку s2 помещаются n символов из строки
                    // начиная с символа номер s
s2.right ( s1, n )  // в строку s2 помещаются n самых правых
                    // символов строки s1
```

Вы можете использовать цикл for для копирования символ за символом подходящих частей строки s1 во временный объект класса Pstring2, который затем их возвратит. Для получения лучшего результата используйте в этих функциях возврат по ссылке, чтобы они могли быть использованы с левой стороны знака «равно» для изменения части существующей строки.

9. Вспомним классы publication, book и type из упражнения 1. Предположим, что мы хотим добавить в классы book и type дату выхода книги. Создайте новый производный класс publication2, который является производным класса publication и включает в себя поле, хранящее эту дату. Затем измените классы book и type так, чтобы они стали производными класса publication2 вместо publication. Сделайте необходимые изменения функций классов так, чтобы пользователь мог вводить и выводить дату выхода книги. Для даты можно использовать класс data из упражнения 5 главы 6, который хранит дату в виде трех целых: для месяца, для дня и для года.

10. В программе EMPMULT этой главы существует только один тип должности менеджера. В любой серьезной компании кроме менеджеров существуют еще и управляющие. Создадим производный от класса manager класс executive. (Мы предполагаем, что управляющий — это главный менеджер.) Добавочными данными этого класса будут размер годовой премии и количество его акций в компании. Добавьте подходящие методы для этих данных, позволяющие их вводить и выводить.

11. В различных ситуациях иногда требуется работать с двумя числами, объединенными в блок. Например, каждая из координат экрана имеет горизонтальную составляющую (x) и вертикальную (y). Представьте такой блок чисел в качестве структуры pair, которая содержит две переменные типа int.

    Теперь предположим, что мы хотим иметь возможность хранить переменные типа pair в стеке. То есть мы хотим иметь возможность положить переменную типа pair в стек путем вызова метода push() с переменной типа pair в качестве аргумента и вынуть ее из стека путем вызова метода pop(), возвращающего переменную типа pair. Начнем с класса Stack2 программы STAKEN из этой главы. Создадим производный от него класс pairStack. В нем будут содержаться два метода: перегружаемый метод push() и перегружаемый метод pop(). Метод pairStack::push() должен будет сделать два вызова метода Stack2::push(), чтобы сохранить оба числа из пары, а метод pairStack::pop() должен будет сделать два вызова метода Stack2::pop().

12. Рассмотрим старую Британскую платежную систему фунты-стерлинги-пенсы (см. упражнение 10 главы 4 «Структуры»). Пенни в дальнейшем делятся на фартинги и полупенни. Фартинг — это 1/4 пенни. Существовали монеты фартинг, полфартинга и пенни. Любые сочетания монет выражались через восьмые части пенни:

   1/8 пенни — это полфартинга;

   1/4 пенни — это фартинг;

   3/8 пенни — это фартинг с половиной;

   1/2 пенни — это полпенни;

   5/8 пенни — это полфартинга плюс полпенни;

   3/4 пенни — это полпенни плюс фартинг;

   7/8 пенни — это полпенни плюс фартинг с половиной.

   Давайте предположим, что мы хотим добавить в класс sterling возможность пользоваться этими дробными частями пенни. Формат ввода/вывода может быть похожим на £1.1.1-1/4 или £9.19.11-7/8, где дефисы отделяют дробные части от пенни.

   Создайте новый класс sterfrac, производный от класса sterling. В нем должна быть возможность выполнения четырех основных арифметических операций со стерлингами, включая восьмые части пенни. Поле eighths типа int определяет количество восьмых. Вам нужно будет перегрузить методы класса sterling, чтобы они могли работать с восьмыми частями. Пользователь должен иметь возможность ввести и вывести дробные части пенни. Не обязательно использовать класс fraction полностью (см. упражнение 11 главы 6), но вы можете это сделать для большей точности.

Глава 10
Указатели

- ✦ Адреса и указатели
- ✦ Операция получения адреса &
- ✦ Указатели и массивы
- ✦ Указатели и функции
- ✦ Указатели на строки
- ✦ Управление памятью: операции new и delete
- ✦ Указатели на объекты
- ✦ Связный список
- ✦ Указатели на указатели
- ✦ Пример разбора строки
- ✦ Симулятор: лошадиные скачки
- ✦ UML-диаграммы
- ✦ Отладка указателей

Для программистов на C++ указатели являются настоящим кошмаром, который заставит растеряться кого угодно. Но бояться не стоит. В этой главе мы попытаемся прояснить тему указателей и рассмотреть их применение в программировании.

Для чего нужны указатели? Вот наиболее частые примеры их использования:

- ✦ доступ к элементам массива;
- ✦ передача аргументов в функцию, от которой требуется изменить эти аргументы;
- ✦ передача в функции массивов и строковых переменных;
- ✦ выделение памяти;
- ✦ создание сложных структур, таких, как связный список.

Указатели — это важная возможность языка C++, поскольку многие другие языки программирования, такие, как Visual Basic или Java, вовсе не имеют указа-

телей. (В Java используются ссылки.) Действительно ли указатели являются столь необходимыми? В предыдущих главах мы видели, что многое можно сделать и без них. Некоторые операции, использующие указатели, могут быть выполнены и другими путями. Например, доступ к элементу массива мы можем получить и не используя указатели (разницу мы рассмотрим позднее), а функция может изменить аргумент, переданный не только по указателю, но и по ссылке.

Однако в некоторых ситуациях указатели являются необходимым инструментом увеличения эффективности программ на языке C++. Замечательным примером является создание таких структур данных, как связные списки или бинарные деревья. Кроме того, некоторые ключевые возможности языка C++, такие, как виртуальные функции, операция new, указатель this (которые мы обсудим в главе 11 «Виртуальные функции»), требуют использования указателей. Поэтому, хотя мы и можем многое сделать без них, указатели будут нам необходимы для более эффективного использования языка программирования.

В этой главе мы познакомимся с указателями, начав с основных концепций и закончив сложными случаями с применением указателей.

Если вы уже знакомы с языком C, то вы можете лишь поверхностно ознакомиться с первой частью главы. Однако вам следует остановиться на разделах второй половины главы, посвященных таким вопросам, как операции new и delete, доступ к функциям классов с использованием указателей, массивы указателей на объекты и связные списки.

Адреса и указатели

Идея указателей несложна. Начать нужно с того, что каждый байт памяти компьютера имеет *адрес*. Адреса — это те же числа, которые мы используем для домов на улице. Числа начинаются с 0, а затем возрастают — 1, 2, 3 и т. д. Если у нас есть 1 Мбайт памяти, то наибольшим адресом будет число 1 048 575 (хотя обычно памяти много больше).

Загружаясь в память, наша программа занимает некоторое количество этих адресов. Это означает, что каждая переменная и каждая функция нашей программы начинается с какого-либо конкретного адреса. На рис. 10.1 показано, как это выглядит.

Операция получения адреса &

Мы можем получить адрес переменной, используя *операцию получения адреса* &. Рассмотрим небольшую программу VARADDR, показывающую, как это сделать:

```
// varaddr.cpp
// адрес переменной
#include <iostream>
using namespace std;

int main ( )
```

```
{
  int var1 = 11;          // определим три переменных
  int var2 = 22;          // и присвоим им некоторые значения
  int var3 = 33;

  cout << &var1 << endl // напечатаем адреса этих переменных
       << &var2 << endl
       << &var3 << endl;

  return 0;
}
```

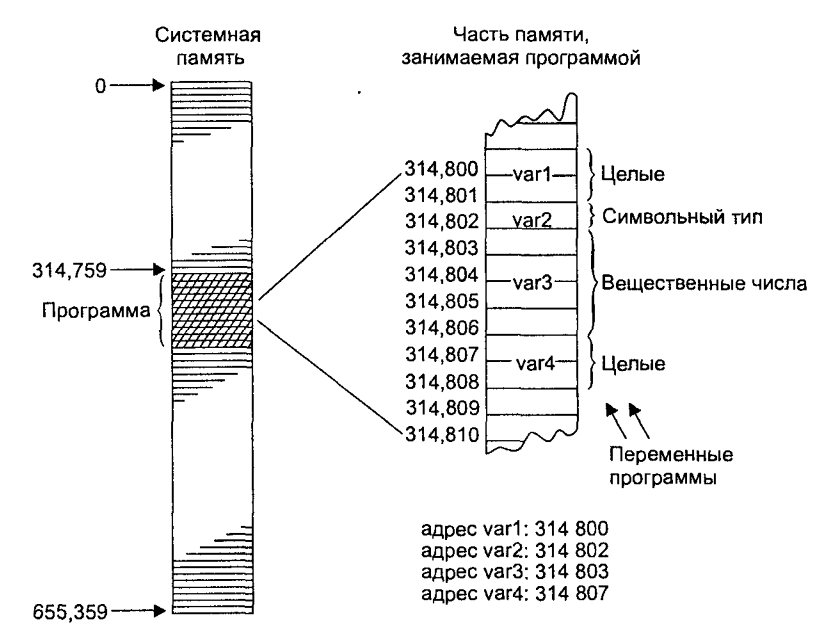


Рис. 10.1. Адреса в памяти

В этой простой программе определены три целочисленные переменные, которые инициализированы значениями 11, 22 и 33. Мы выводим на экран адреса этих переменных.

Реальные адреса, занятые переменными в программе, зависят от многих факторов, таких, как компьютер, на котором запущена программа, размер оперативной памяти, наличие другой программы в памяти и т. д. По этой причине вы, скорее всего, получите совершенно другие адреса при запуске этой программы (вы даже можете не получить одинаковые адреса, запустив программу несколько раз подряд). Вот что мы получили на нашей машине:

```
0x8f4ffff4- адрес переменной var1
0x8f4ffff2- адрес переменной var2
0x8f4ffff0- адрес переменной var3
```

Запомните, что *адреса* переменных — это не то же самое, что их *значение*. Содержимое трех переменных — это 11, 22 и 33. На рис. 10.2 показано размещение трех переменных в памяти.

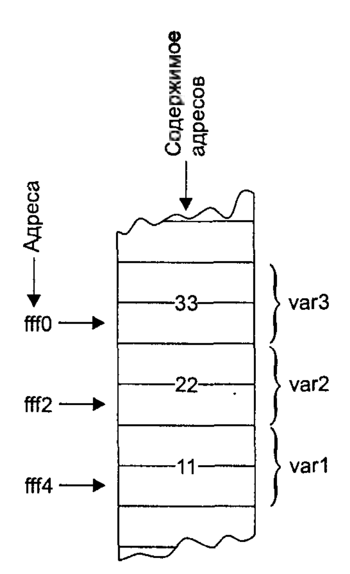


Рис. 10.2. Адреса и содержимое переменных

Использование операции << позволяет показать адреса в шестнадцатеричном представлении, что видно из наличия префикса 0x перед каждым числом. Это обычная форма записи адресов памяти. Не беспокойтесь, если вы не знакомы с шестнадцатеричной системой. Вам всего лишь нужно помнить, что каждая переменная имеет уникальный адрес. Однако вы могли заметить в выводе на дисплей, что адреса отличаются друг от друга двумя байтами. Это произошло потому, что переменная типа int занимает два байта в памяти (в 16-битной системе). Если бы мы использовали переменные типа char, то значения адресов отличались бы на единицу, так как переменные занимали бы по 1 байту, а если бы мы использовали тип double, то адреса отличались бы на 8 байтов.

Адреса расположены в убывающем порядке, потому что локальные переменные хранятся в стеке, где адреса располагаются по убыванию. Если же использовать глобальные переменные, то их адреса располагаются в порядке возрастания, так как глобальные переменные хранятся в куче, где адреса располагаются по возрастанию. Но вас не должны сильно волновать эти детали, так как компилятор скрывает от вас такие подробности.

Не путайте операцию адреса переменной, стоящую перед ее именем, и операцию ссылки, стоящую за именем типа в определении или прототипе функции (ссылки мы обсуждали в главе 5 «Функции»).

Переменные указатели

Адресное пространство ограничено. Возможность узнать, где расположены в памяти переменные, полезна (мы делали это в программе VARADDR), а видеть само значение адреса нам нужно не всегда. Потенциальное увеличение наших возможностей в программировании требует реализации следующей идеи: нам необходимы *переменные, хранящие значение адреса*. Нам знакомы переменные, хранящие

знаки, числа с плавающей точкой, целые и т. д. Адреса хранятся точно так же. Переменная, содержащая в себе значение адреса, называется *переменной-указателем* или просто *указателем*.

Какого же типа может быть переменная-указатель? Она не того же типа, что и переменная, адрес которой мы храним: указатель на int не имеет типа int. Возможно, вы думаете, что указатели принадлежат некоторому типу pointer или ptr. Однако здесь все немного сложнее. В программе PTRVAR показан синтаксис переменных-указателей.

```
// ptrvar.cpp
// указатели (адреса переменных)
#include <iostream>
using namespace std;

int main ( )
{
  int var1 = 11;        // две переменные
  int var2 = 22;

  cout << &var1 << endl // покажем адреса переменных
       << &var2 << endl << endl;

  int* ptr; // это переменная-указатель на целое

  ptr = &var1;          // присвоим ей значение адреса var1
  cout << ptr << endl;  // и покажем на экране

  ptr = &var2;          // теперь значение адреса var2
  cout << ptr << endl;  // и покажем на экране

  return 0;
}
```

В этой программе определены две целочисленных переменных var1 и var2, которые инициализированы значениями 11 и 22. Затем программа выводит на дисплей их адреса.

Далее в программе определена *переменная-указатель* в строке

```
int* ptr;
```

Для непосвященных это достаточно странный синтаксис. Звездочка означает *указатель на*. Таким образом, в этой строке определена переменная ptr как *указатель на* int, то есть эта переменная может содержать в себе адрес переменной типа int.

Почему же идея создания типа pointer, который бы включал в себя указатели на данные любого типа, оказалась неудачной? Если бы этот тип существовал, то мы могли бы записать объявление переменной этого типа как:

```
pointer ptr;
```

Проблема в том, что компилятору нужны сведения о том, *какого именно типа переменная, на которую указывает указатель* (мы поймем почему, когда будем рассматривать указатели и массивы). Синтаксис, используемый в C++, позволяет нам объявить указатель следующим образом:

```
char*     cptr:    // указатель на символьную переменную
int*      iptr:    // указатель на целую переменную
float*    fptr:    // указатель на вещественную переменную
Distance* distptr: // указатель на переменную класса Distance
```

Недостатки синтаксиса

Нужно заметить, что общепринято определение указателя с помощью звездочки, записываемой перед именем переменной, а не сразу после названия типа.

```
char *charptr:
```

Это не принципиально для компилятора, но звездочка, расположенная сразу за названием типа переменной, сигнализирует о том, что это не просто тип, а указатель на него.

Если мы определяем в одной строке более чем один указатель одного и того же типа, то звездочку необходимо ставить перед именем каждой переменной.

```
char* ptr1. * ptr2. * ptr3: // три переменных указателя
```

И в таком случае можно использовать стиль написания, при котором звездочка ставится рядом с именем:

```
char *ptr1. *ptr2. *ptr3:
```

Указатели должны иметь значение

Пусть у нас есть адрес 0x8f4ffff4, мы можем его назвать *значением указателя*. Указатель ptr называется *переменной-указателем*. Как переменная var1 типа int может принимать значение, равное 11, так переменная-указатель может принимать значение, равное 0x8f4ffff4.

Пусть мы определили переменную для хранения некоторого значения (пока мы ее не инициализировали). Она будет содержать некое случайное число. В случае с переменной-указателем это случайное число является неким адресом в памяти. Перед использованием указателя необходимо инициализировать его определенным адресом. В программе PTRVAR указателю ptr присваивается адрес переменной var1:

```
ptr = &var1: // помещает в переменную ptr адрес переменной var1
```

Затем программа выводит на дисплей значение, содержащееся в переменной ptr, это будет адрес переменной var1. Далее указатель принимает значение адреса переменной var2 и выводит его на экран. На рис. 10.3 показаны действия программы PTRVAR, а ниже мы приведем результат ее работы:

```
0x8f51fff4- адрес переменной var1
0x8f51fff2- адрес переменной var2

0x8f51fff4- значение ptr равно адресу переменной var1
0x8f51fff2- значение ptr равно адресу переменной var2
```

Подведем итог. Указатель может хранить адрес переменной соответствующего типа. Ему должно быть обязательно присвоено некоторое значение, иначе

случайный адрес, на который он указывает, может оказаться чем угодно: от кода нашей программы до кода операционной системы. Неинициализированный указатель может привести к краху системы, его очень тяжело выявить при отладке программы, так как компилятор не выдает предупреждения о подобных ошибках. Поэтому всегда важно убедиться, что каждому указателю перед его использованием было присвоено значение.

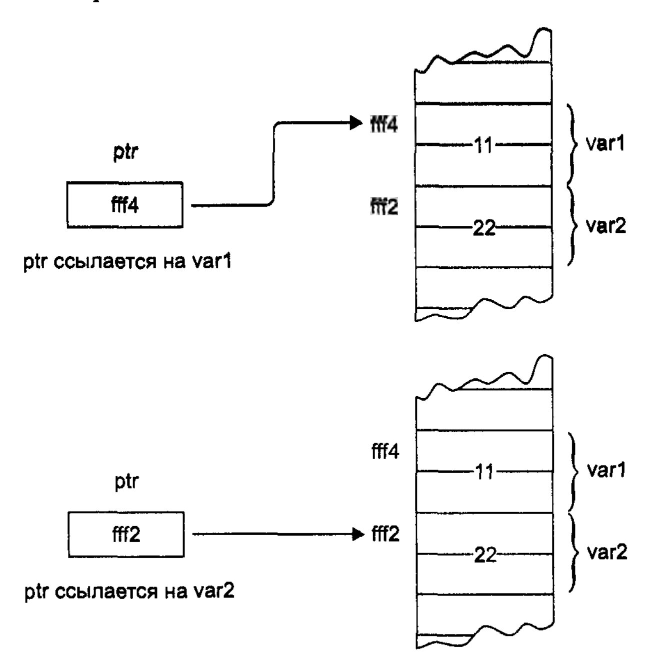


Рис. 10.3. Изменение значений указателя ptr

Доступ к переменной по указателю

Предположим, что мы не знаем имени переменной, но знаем ее адрес. Можем ли мы получить доступ к значению этой переменной? (Возможно, мы потеряли список имен переменных.)

Существует специальная запись, предназначенная для доступа к значению переменной, используя ее адрес вместо имени. В программе PTRACC показано, как это можно сделать:

```
// ptracc.cpp
// доступ к переменной через указатель
#include <iostream>
using namespace std;

int main ( )
{
  int var1 = 11;        // две переменные
  int var2 = 22;
```

```
  int* ptr;             // указатель на целое

  ptr = &var1;          // помещаем в ptr адрес переменной var1
  cout << *ptr << endl; // показываем содержимое переменной через указатель

ptr = &var2;            // помещаем в ptr адрес переменной var2
  cout << *ptr << endl; // показываем содержимое переменной через указатель

  return 0;
}
```

Эта программа очень похожа на PTRVAR, за исключением того, что вместо вывода на дисплей адресов, хранящихся в переменной ptr, мы выводим значения, хранящиеся по адресу, на который указывает ptr. Результат работы программы будет следующим:

```
11
22
```

Выражение *ptr, дважды встречающееся в нашей программе, позволяет нам получить значения переменных var1 и var2.

Звездочка, стоящая перед именем переменной, как в выражении *ptr, называется *операцией разыменования*. Эта запись означает: взять *значение переменной, на которую указывает указатель*. Таким образом, выражение *ptr представляет собой значение переменной, на которую указывает указатель ptr. Если ptr указывает на переменную var1, то значением выражения *ptr будет 11. На рис. 10.4 показано, как работает операция разыменования.

Указатели можно использовать не только для получения значения переменной, на которую он указывает, но и для выполнения действий с этими переменными. Программа PTRTO использует указатель для присвоения значения одной переменной, а затем другой:

```
// ptrto.cpp
// еще один пример доступа через указатель
#include <iostream>
using namespace std;

int main ( )
{
  int var1, var2;       // две переменные
  int* ptr;             // указатель на целое

  ptr = &var1;          // пусть ptr указывает на var1
  *ptr = 37;            // то же самое, что var1 = 37;
  var2 = *ptr;          // то же самое, что var2 = var1;

  cout << var2 << endl; // убедимся, что var2 равно 37

  return 0;
}
```

Запомните, что звездочка, используемая в операции разыменования, — это не то же самое, что звездочка, используемая при объявлении указателя. Операция

разыменования *предшествует* имени переменной и означает *значение, находящееся в переменной, на которую указывает указатель*. Звездочка же в объявлении указателя означает *указатель на*.

```
int* ptr;  // объявление: указатель на int
*ptr = 37; // разыменование: значение переменной, адресованной через ptr
```

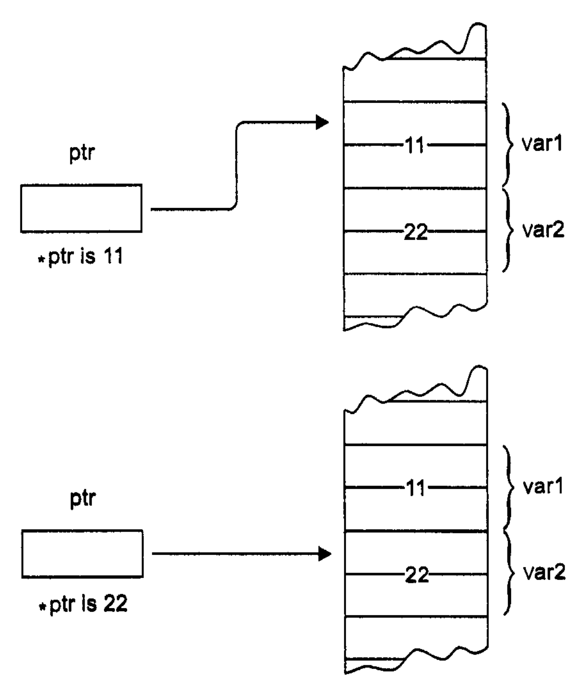


Рис. 10.4. Доступ к переменным через указатели

Доступ к значению переменной, хранящейся по адресу, с использованием операции разыменования называется *непрямым доступом* или *разыменованием указателя*.

Рассмотрим небольшие примеры того, что мы изучили:

```
int v;   // определим переменную v типа int
int* p;  // определим переменную типа указатель на int
p = &v;  // присвоим переменной p значение адреса переменной v
v = 3;   // присвоим v значение 3
*p = 3;  // сделаем то же самое, но через указатель
```

В последних двух строках показано различие между прямым доступом, когда мы обращаемся к переменной, используя ее имя, и непрямым доступом, когда мы обращаемся к той же переменной, но уже используя ее адрес.

Пока в примерах программ мы не обнаружили преимущества использования указателей для доступа к переменным, поскольку мы всегда можем применить прямой доступ. Важность указателей становится очевидной в том случае, когда мы не имеем прямого доступа к переменной. Мы увидим это позже.

Указатель на void

Перед тем как мы продолжим рассматривать работу указателей, необходимо отметить одну их особенность. Адрес, который помещается в указатель, должен быть одного с ним типа. Мы не можем присвоить указателю на int адрес переменной типа float.

```
float flovar = 98.6;
int* ptrint = &flovar; // Так нельзя: типы int* и float* несовместимы
```

Однако есть одно исключение. Существует тип указателя, который может указывать на любой тип данных. Он называется указателем на void и определяется следующим образом:

```
void* ptr; // указатель, который может указывать на любой тип данных
```

Такие указатели предназначены для использования в определенных случаях, например передача указателей в функции, которые работают независимо от типа данных, на который указывает указатель.

Рассмотрим пример использования указателя на void. В этой программе также показано, что если вы не используете указатель на void, то вам нужно быть осторожными, присваивая указателю адрес того же типа, что и указатель. Вот листинг программы PTRVOID[1]:

```
// ptrvoid.cpp
// указатель на void
#include <iostream>
using namespace std;

int main ( )
{
  int intvar;            // целочисленная переменная
  float flovar;          // вещественная переменная

  int* ptrint;           // указатель на int
  float* ptrflo;         // указатель на float
  void* ptrvoid;         // указатель на void

  ptrint = &intvar;      // так можно: int* = int*
//ptrint = &flovar;      // так нельзя: int* = float*
//ptrflo = &intvar;      // так нельзя: float* = int*
  ptrflo = &flovar;      // так можно: float* = float*

  ptrvoid = &intvar;     // так можно: void* = int*
  ptrvoid = &flovar;     // так можно: void* = float*

  return 0;
}
```

Мы можем присвоить адрес intvar переменной ptrint, потому что обе этих переменных имеют тип int*. Присвоить же адрес flovar переменной ptrint мы не можем, поскольку они разных типов: переменная flovar типа float*, а переменная

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

ptrint типа int*. Однако указателю ptrvoid может быть присвоено значение любого типа, так как он является указателем на void.

Если вам по некоторым причинам необходимо присвоить одному типу указателя другой тип, то вы можете использовать функцию reinterpret_cast. Для строк программы PTRVOID, которые мы закомментировали, это будет выглядеть следующим образом:

```
ptrint = reinterpret_cast<int*>( &flovar );
ptrflo = reinterpret_cast<float*>( &intvar );
```

Использование функции reinterpret_cast в этом случае нежелательно, но может оказаться выходом из сложной ситуации. Нединамические вычисления не работают с указателями. В старой версии С с вычислениями можно было так поступать, но для С++ это будет плохим тоном. Мы рассмотрим примеры функции reinterpret_cast в главе 12 «Потоки и файлы», где она используется для изменения способа интерпретации данных из буфера.

Указатели и массивы

Указатели и массивы очень похожи. В главе 7 «Массивы и строки» мы рассмотрели, как можно получить доступ к элементам массива. Вспомним это на примере ARRNOTE:

```
// arrnote.cpp
// обычный доступ к элементам массива
#include <iostream>
using namespace std;

int main ( )
{
  int intarray [ 5 ] = { 31, 54, 77, 52, 93 }; // набор целых чисел

  for ( int j = 0; j < 5; j++ )      // для каждого элемента массива
    cout << intarray [ j ] << endl; // напечатаем его значение

  return 0;
}
```

Функция cout выводит элементы массива по очереди. Например, при j, равном 3, выражение intarray[j] принимает значение intarray[3], получая доступ к четвертому элементу массива, числу 52. Рассмотрим результат работы программы ARRNOTE:

```
31
54
77
52
93
```

Необычно то, что доступ к элементам массива можно получить как используя операции с массивами, так и используя указатели. Следующий пример PTRNOTE похож на пример ARRNOTE, за исключением того, что в нем используются указатели.

```
// ptrnote.cpp
// доступ к элементам массива через указатель
#include <iostream.h>
using namespace std;

int main ( )
{
  int intarray [ 5 ] = { 31, 54, 77, 52, 93 }; // набор целых чисел

  for ( int j = 0; j < 5; j++ )        // для каждого элемента массива
    cout << *( intarray + j ) << endl; // напечатаем его значение

  return 0;
}
```

Результат действия выражения *(intarray+j) — тот же, что и выражения intarray[j] в программе ARRNOTE, при этом результат работы программ одинаков. Что же представляет из себя выражение *(intarray+j)? Допустим, j равно 3, тогда это выражение превратится в *(intarray+3). Мы предполагаем, что оно содержит в себе значение четвертого элемента массива (52). Вспомним, что имя массива является его адресом. Таким образом, выражение intarray+j — это адрес чего-то в массиве. Вы можете ожидать, что intarray+3 будет означать 3 байта массива intarray. Но это не даст нам результат, который мы хотим получить: intarray — это массив элементов типа int, и три байта в этом массиве — середина второго элемента, что не очень полезно для нас. Мы хотим получить четвертый *элемент* массива, а не его четвертый *байт*, что показано на рис. 10.5 (на этом рисунке int занимает 2 байта).

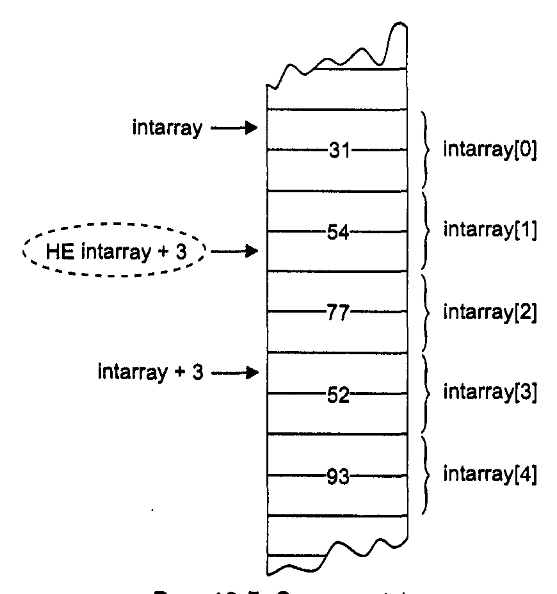


Рис. 10.5. Отсчет по int

Компилятору C++ достаточно получить размер данных в счетчике для выполнения вычислений с адресами данных. Ему известно, что intarray — массив

типа int. Поэтому, видя выражение intarray+3, компилятор интерпретирует его как адрес четвертого *элемента* массива, а не четвертого байта.

Но нам необходимо значение четвертого элемента, а не его *адрес*. Для его получения мы используем операцию разыменования (*). Поэтому результатом выражения *(intarray+3) будет значение четвертого элемента массива, то есть 52.

Теперь рассмотрим, почему при объявлении указателя мы должны указать тип переменной, на которую он будет указывать. Компилятору необходимо знать, на переменные какого типа указывает указатель, чтобы осуществлять правильный доступ к элементам массива. Он умножает значение индекса на 2, в случае типа int, или на 8, в случае типа double.

Указатели-константы и указатели-переменные

Предположим, что мы хотим использовать операцию увеличения вместо прибавления шага j к имени intarray. Можем ли мы записать *(intarray++)?

Сделать так мы не можем, поскольку нельзя изменять константы. Выражение intarray является адресом в памяти, где ваш массив будет храниться до окончания работы программы, поэтому intarray — это указатель константы. Мы не можем сказать intarray++, так же как не можем сказать 7++. (В многозадачных системах адресная переменная может менять свое значение в течение выполнения программы. Активная программа может обмениваться данными с диском, а затем снова загружать их уже в другие участки памяти. Однако этот процесс невидим в нашей программе.)

Мы не можем увеличивать адрес, но можем увеличить указатель, который содержит этот адрес. В примере PTRINC мы покажем, как это работает:

```
// ptrinc.cpp
// доступ к массиву через указатель
#include <iostream>
using namespace std;

int main ( )
{
  int intarray [ 5 ] = { 31, 54, 77, 52, 93 }; // набор целых чисел
  int* ptrint;                     // указатель на int
  ptrint = intarray;               // пусть он указывает на наш массив

  for( int j = 0; j < 5; j++ )     // для каждого элемента массива
    cout << *( ptrint++ ) << endl; // напечатаем его значение

  return 0;
}
```

Здесь мы определили указатель на int — ptrint — и затем присвоили ему значение адреса массива intarray. Теперь мы можем получить доступ к элементам массива, используя выражение

```
*( ptrint++ )
```

Переменная ptrint имеет тот же адрес, что и intarray, поэтому доступ к первому элементу массива intarray[0], значением которого является 31, мы можем осуществлять, как и раньше. Но так как переменная ptrint не является константой, то мы можем ее увеличивать. После увеличения она уже будет показывать на второй элемент массива intarray[1]. Значение этого элемента массива мы можем получить, используя выражение *(ptrint++). Продолжая увеличивать ptrint, мы можем получить доступ к каждому из элементов массива по очереди. Результат работы программы PTRINC будет тем же, что и программы PTRNOTE.

Указатели и функции

В главе 5 мы упомянули, что передача аргументов функции может быть произведена тремя путями: по значению, по ссылке и по указателю. Если функция предназначена для изменения переменной в вызываемой программе, то эта переменная не может быть передана по значению, так как функция получает только копию переменной. Однако в этой ситуации мы можем использовать передачу переменной по ссылке и по указателю.

Передача простой переменной

Сначала мы рассмотрим передачу аргумента по ссылке, а затем сравним ее с передачей по указателю. В программе PASSREF рассмотрена передача по ссылке.

```
// passref.cpp
// передача аргумента по ссылке
#include <iostream>
using namespace std;

int main ( )
{
  void centimize ( double& ); // прототип функции

  double var = 10.0;          // значение переменной var равно 10 (дюймов)
  cout << "var = " << var << "дюймов" << endl;

  centimize ( var );          // переведем дюймы в сантиметры
  cout << "var= " << var << "сантиметров" << endl;

  return 0;
}
///////////////////////////////////////////////////////////
void centimize ( double& v )
{
  v *= 2.54; // v – это то же самое, что и var
}
```

В этом примере мы хотим преобразовать значение переменной var функции main() из дюймов в сантиметры. Мы передаем переменную по ссылке в функцию centimize(). (Помним, что &, следующий за типом double в прототипе этой функции, означает, что аргумент передается по ссылке.) Функция centimize()

умножает первоначальное значение переменной на 2,54. Обратим внимание, как функция ссылается на переменную. Она просто использует имя аргумента v; v и var — это различные имена одного и того же.

После преобразования var в сантиметры функция main() выведет полученный результат. Итог работы программы PASSREF:

```
var = 10 дюймов
var = 25.4 сантиметров
```

В следующем примере PASSPTR мы рассмотрим, как в аналогичной ситуации используются указатели.

```
// passptr.cpp
// передача аргумента по указателю
#include <iostream>
using namespace std:

int main ( )
{
  void centimize ( double* ); // прототип функции

  double var = 10.0;          // значение переменной var равно 10 (дюймов)
  cout << "var = " << var << "дюймов" << endl;

  centimize ( &var ):         // переведем дюймы в сантиметры
  cout << "var= " << var << "сантиметров" << endl;

  return 0:
}
//////////////////////////////////////////////////////////////
void centimize ( double* ptrd )
{
  *ptrd *= 2.54;              // *ptrd - это то же самое, что и var
}
```

Результаты работы программ PASSPTR и PASSREF одинаковы.

Функция centimize() объявлена использующей параметр как указатель на double:

```
void centimize ( double* ): // аргумент - указатель на double
```

Когда функция main() вызывает функцию centimize(), она передает в качестве аргумента адрес переменной:

```
centimize ( &var );
```

Помним, что это не сама переменная, как это происходит при передаче по ссылке, а ее адрес.

Так как функция centimize() получает адрес, то она может использовать операцию разыменования *ptrd для доступа к значению, расположенному по этому адресу:

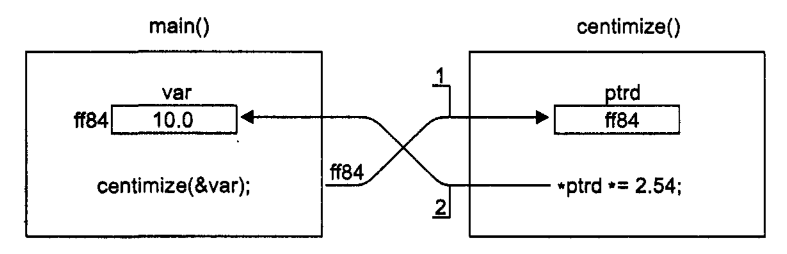
```
*ptrd *= 2.54;     // умножаем содержимое переменной по адресу ptrd на 2.54
```

Это то же самое, что

```
*ptrd = *ptrd * 2.54:
```

где одиночная звездочка означает умножение (эта операция действительно распространена).

Так как ptrd содержит адрес переменной var, то все действия, которые мы совершим с *ptrd, мы в действительности проделаем с var. На рис. 10.6 показано, как изменение *ptrd в функции изменяет переменную var в вызывающей программе.



1. main() передает адрес переменной var в ptrd в centimize()

2. centimize() использует этот адрес для доступа к var

Рис. 10.6. Передача в функцию по указателю

Передача указателя в функцию в качестве аргумента в некоторых случаях похожа на передачу по ссылке. Они обе позволяют переменной вызывающей программы быть измененной в функции. Однако их механизмы различны. Ссылка — это псевдоним переменной, а указатель — это адрес переменной.

Передача массивов

Начиная с главы 7, мы рассматривали многочисленные примеры, в которых массивы передаются как аргументы в функции, при этом функция имела доступ к их элементам. До этой главы мы использовали только операции с массивами, так как нам были неизвестны указатели. Однако при передаче массива в функцию принято использовать указатели, а не просто операции массива. Рассмотрим это в программе PASSARR:

```
// passarr.cpp
// передача массива по указателю
#include <iostream>
using namespace std;
const int MAX = 5;            // количество элементов в массиве

int main ( )
{
  void centimize ( double* ); // прототип функции

  double varray [ MAX ] = { 10.0, 43.1, 95.9, 58.7, 87.3 };

  centimize( varray );        // переводим все элементы массива в сантиметры

  // покажем, что у нас получилось
  for ( int j = 0; j < MAX; j++ )
```

```
    cout << "varray [ " << j << " ] = " << varray [ j ] << " сантиметров" << endl;

  return 0;
}
////////////////////////////////////////////////////////////
void centimize ( double* ptrd )
{
  for ( int j = 0; j < MAX; j++ )
    *ptrd++ *= 2.54;
}
```

Прототип функции тот же, что и в программе PASSPTR; функция имеет один аргумент, указатель на double. При передаче массива по значению это выглядело бы так:

```
void centimize ( double[ ] );
```

Записи double* и double [ ] эквивалентны, но синтаксис указателей используется чаще.

Так как имя массива является его адресом, то нам нет надобности использовать операцию взятия адреса & при вызове функции:

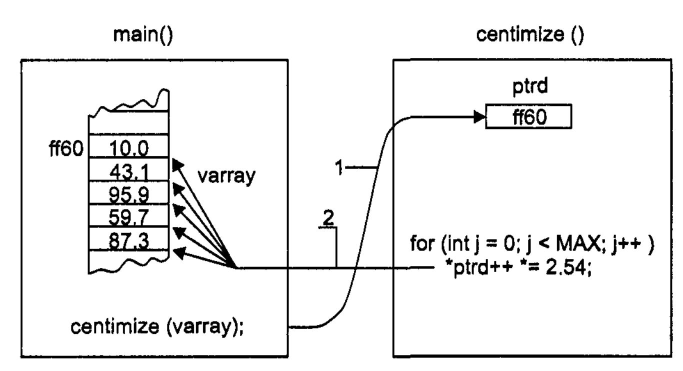
```
centimize ( array );    // передается адрес массива
```

В функции centimize() адрес массива присваивается переменной ptrd. Мы просто используем операцию увеличения для указателя ptrd, чтобы указать на все элементы массива по очереди:

```
*ptrd++ *= 2.54;
```

На рис. 10.7 показан доступ к массиву. А результат работы программы PASSARR будет следующим:

```
varray [ 0 ] = 25.4 сантиметров
varray [ 1 ] = 109.474 сантиметров
varray [ 2 ] = 243.586 сантиметров
varray [ 3 ] = 151.638 сантиметров
varray [ 4 ] = 221.742 сантиметров
```



1. main() передает адрес varray в ptrd функции centimize()
2. centimize() использует этот адрес для доступа ко всем элементам массива по очереди

Рис. 10.7. Доступ к массиву из функции

Рассмотрим вопрос синтаксиса: как узнать, что в выражении *ptrd++ увеличивается указатель, а не его содержимое? Другими словами, как компилятор интерпретирует это выражение: как *(ptrd++), что нам и нужно, или как (*ptrd)++? Здесь * (при использовании в качестве операции разыменования) и ++ имеют одинаковый приоритет. Однако операции одинакового приоритета различаются еще и другим способом: *ассоциативностью*. Ассоциативность определяет, как компилятор начнет выполнять операции, справа или слева. В группе операций, имеющих правую ассоциативность, компилятор выполняет сначала операцию, стоящую справа. Унарные операции * и ++ имеют правую ассоциативность, поэтому наше выражение интерпретируется как *(ptrd++) и увеличивает указатель, а не то, на что он указывает. Таким образом, сначала увеличивается указатель, а затем к результату применяется операция разыменования.

Сортировка элементов массива

Давайте рассмотрим сортировку содержимого массива в качестве примера использования указателей для доступа к элементам массива. Мы рассмотрим две программы. В первой заложим основу, а во второй (являющейся продолжением первой) продемонстрируем процесс сортировки.

Расстановка с использованием указателей

Первая программа похожа на программу REFORDER из главы 6 «Объекты и классы» за исключением того, что здесь вместо ссылок используются указатели. Она упорядочивает два числа, которые передаются как аргументы, меняя их местами, если второе меньше первого. Вот листинг программы PTRORDER:

```
// ptrorder.cpp
// сортировка двух аргументов по указателю
#include <iostream>
using namespace std;

int main ( )
{
  void order ( int*, int* );     // прототип функции

  int n1 = 99, n2 = 11;
  int n3 = 22, n4 = 88;

  order ( &n1, &n2 );
  order ( &n3, &n4 );

  cout << "n1 = " << n1 << endl; // выводим переменные
  cout << "n2 = " << n2 << endl; // на экран
  cout << "n3 = " << n3 << endl;
  cout << "n4 = " << n4 << endl;

  return 0;
}
//////////////////////////////////////////////////////////
void order ( int* numb1, int* numb2 ) // сортировка двух чисел
```

```
{
  if ( *numb1 > *numb2 )              // если первое число больше, то меняем их местами
  {
    int temp = *numb1;
    numb1 = numb2;
    numb2 = temp;
  }
}
```

Функция order() работает так же, как и в программе REFORDER, за исключением того, что здесь передаются адреса сортируемых чисел и доступ к числам происходит через указатели. Так, *numb1 получает значение переменной n1 в функции main() и является первым аргументом, а *numb2 получает значение переменной n2.

Приведем результат работы программы PTRORDER:

```
n1 = 11
n2 = 99
n3 = 22
n4 = 88
```

Мы будем использовать функцию order() из программы PTRORDER в нашей следующей программе, PTRSORT, которая сортирует массив целых чисел.

```
// ptrsort.cpp
// сортировка массива с использованием указателей
#include <iostream>
using namespace std;

int main ( )
{
  void bsort ( int*, int ); // прототип функции
  const int N = 10;         // размер массива
  int arr [ N ] = { 37, 84, 62, 91, 11, 65, 57, 28, 19, 49 }; // массив для сортировки

  bsort ( arr, N );

  for ( int j = 0; j < N; j++ )
    cout << arr [ j ] << " ";
  cout << endl;

  return 0;
}
///////////////////////////////////////////////////////////
void bsort ( int* ptr, int n )
{
  void order ( int*, int* );          // прототип функции
  int j, k;                           // индексы в нашем массиве

  for ( int j = 0; j < n - 1; j++ )   // внешний цикл
    for ( int k = j + 1; k < n; k++ ) // внутренний цикл
      order ( ptr + j, ptr + k );     // сортируем элементы
}
///////////////////////////////////////////////////////////
void order ( int* numb1, int* numb2 ) // сортировка двух чисел
{
```

```
  if ( *numb1 > *numb2 )            // если первое число
                                    // больше, то меняем их местами
  {
    int temp = *numb1;
    numb1 = numb2;
    numb2 = temp;
  }
}
```

Массив arr целых чисел инициализирован в функции main() неотсортированными значениями. Адрес массива и номера его элементов передаются в функцию bsort(). Она сортирует массив и выводит результат:

```
11 19 28 37 49 57 62 65 84 91
```

Сортировка методом пузырька

Функция bsort() при сортировке массива использует вариацию метода пузырька. Это простая сортировка (хотя довольно медленная). Рассмотрим ее алгоритм. Допустим, что мы хотим расставить элементы массива в порядке возрастания. Сначала первый элемент массива (arr [0]) сравнивается по очереди с остальными элементами (начиная со второго). Если он больше, чем какой-либо из элементов массива, то эти элементы меняются местами. Когда это будет проделано, то нам будет известен первый элемент последовательности, самый маленький. Затем мы сравниваем второй элемент по очереди со всеми остальными, начиная с третьего, и вновь меняем местами элементы, если найдется элемент в массиве меньший, чем второй. После этих действий нам будет уже известен второй элемент последовательности, второй по величине. Этот процесс продолжается далее для всех остальных элементов до предпоследнего, затем массив будет считаться упорядоченным. На рис. 10.8 показана сортировка методом пузырька в действии (с несколькими элементами из PTRSORT).

В PTRSORT на первом месте стоит число 37, оно сравнивается с каждым элементом по очереди и меняется местами с числом 11. На втором месте стоит число 84, которое тоже сравнивается со всеми элементами. Оно меняется местами с числом 62, затем число 62 (которое оказалось теперь на втором месте) меняется местами с числом 37, 37 меняется местами с 28, а 28 с 19. На третьем месте вновь оказывается число 84, оно меняется с 62, которое меняется с 57, затем 57 с 37 и 37 с 28. Процесс продолжается до тех пор, пока массив не будет отсортирован.

Функция bsort() состоит из двух вложенных циклов, каждый из которых контролирует указатель. Внешний цикл использует переменную j, внутренний — переменную k. Выражения ptr+j и ptr+k указывают на различные элементы массива, которые определяются переменными цикла. Выражение ptr+j используется для перемещения по массиву, начиная с первого элемента до предпоследнего. Для каждой позиции, на которую указывает ptr+j во внешнем цикле, выражение ptr+k внутреннего цикла перемещается, начиная со следующего элемента и до конца массива. Элементы внешнего и внутреннего цикла ptr+j и ptr+k сравнива-

ются с использованием функции order(), и если первый больше, чем второй, то они меняются местами. На рис. 10.9 показан этот процесс.

Рис. 10.8. Действия при сортировке методом пузырька

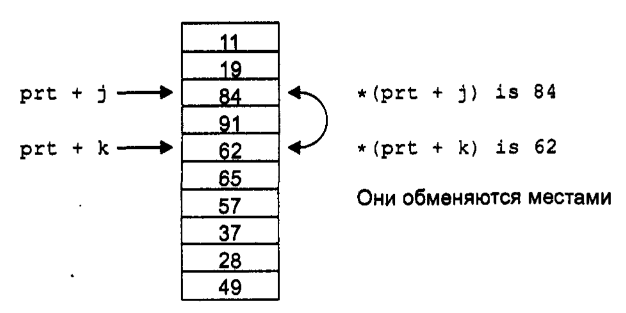


Рис. 10.9. Действия программы PTRSORT

Пример PTRSORT позволяет оценить механизм указателей. Они предоставляют возможность последовательно и эффективно работать с элементами мас-

сива и другими переменными, имена которых нам не известны в определенной функции.

Указатели на строки

Как мы заметили в главе 7, строки — это просто массивы элементов типа char. Таким образом, доступ через указатели может быть применен к элементам строки так же, как и к элементам массива.

Указатели на строковые константы

Рассмотрим программу TWOSTR, в которой определены две строки: одна — с использованием операций массива, а другая — с использованием указателей.

```
// twostr.cpp
// описание строк через массивы и через указатели
#include <iostream>
using namespace std;

int main ( )
{
  char str1 [ ] = "Определение через массив";
  char* str2 = "Определение через указатель";

  cout << str1 << endl; // покажем наши строки
  cout << str2 << endl;

//str1++;               // так делать нельзя
  str2++;               // а так можно

  cout << str2 << endl; // значение str2 немного изменилось

  return 0;
}
```

Во многих случаях эти два способа определения строк эквивалентны. Вы можете распечатать обе строки, используя их как аргументы функций. Это показано в примере. Но есть небольшое различие: str1 — это адрес, то есть указатель-константа, a str2 — указатель-переменная. Поэтому str2 может изменять свое значение, а str1 нет, что показано в программе. Рисунок 10.10 показывает расположение этих двух строк в памяти.

Мы можем увеличить str2, так как это указатель, но после этой операции он уже больше не будет показывать на первый элемент строки. Рассмотрим результат работы программы TWOSTR:

```
Определение через массив
Определение через указатель
Определение через указатель
```

Строка, определенная как указатель, гораздо более гибка, чем строка, определенная как массив. В следующем примере мы используем эту гибкость.

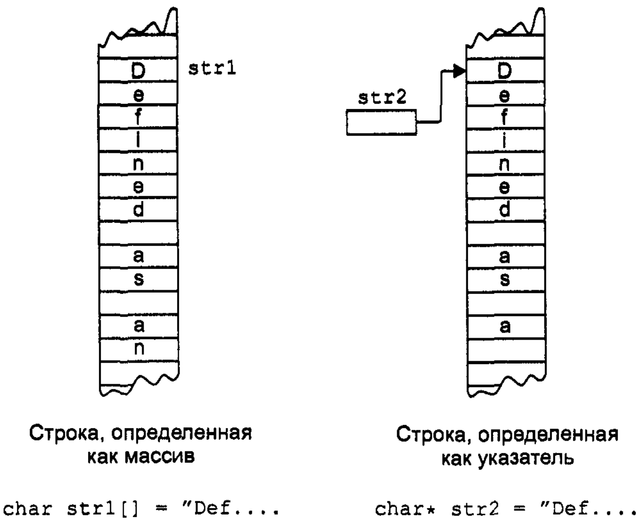


Рис. 10.10. Строки как массив и как указатель

Строки как аргументы функций

Рассмотрим пример, показывающий, как строки используются в качестве аргументов. Функция печатает строку, выводя знаки по очереди. Вот листинг программы PTRSTR:

```
// ptrstr.cpp
// показ строки, определенной через указатель
#include <iostream>
using namespace std;

int main ( )
{
  void dispstr ( char* ); // прототип функции
  char str [ ] = "У бездельников всегда есть свободное время.";

  dispstr ( str );

  return 0;
}
//////////////////////////////////////////////////////////
void dispstr ( char* ps )
{
  while( *ps )             // пока не встретим конец строки
    cout << *ps++;         // будем посимвольно выводить ее на экран
  cout << endl;
}
```

Адрес массива str использован как аргумент при вызове функции dispstr(). Этот адрес является константой, но так как он передается по значению, то в функции dispstr() создается его копия. Это будет указатель ps. Он может быть изменен, и функция увеличивает его, выводя строку на дисплей. Выражение *ps++

возвращает следующий знак строки. Цикл повторяется до появления знака конца строки ('\0'). Так как он имеет значение 0, которое интерпретируется как false, то в этот момент цикл заканчивается.

Копирование строк с использованием указателей

В рассмотренных нами примерах указатели использовались для получения значений элементов массива. Указатели можно также использовать для вставки значений в массив. В следующем примере COPYSTR мы рассмотрим функцию, которая копирует одну строку в другую:

```
// copystr.cpp
// копирует одну строку в другую
#include <iostream>
using namespace std;

int main ( )
{
  void copystr ( char*, const char* ); // прототип функции
  char* str1 = "Поспешишь - людей насмешишь!";
  char str2 [ 80 ];                    // пустая строка

  copystr ( str2, str1 );              // копируем строку str1 в str2
  cout << str2 << endl;                // и показываем результат

  return 0;
}
///////////////////////////////////////////////////////////
void copystr ( char* dest, const char* str )
{
  while ( *src )       // пока не встретим конец строки
    *dest++ = *src++;  // копируем ее
  *dest = '\0';        // заканчиваем строку
}
```

Здесь функция main() вызывает функцию copystr(), которая копирует str1 в str2.

```
*dest++ = *src++;
```

Значение src помещается по адресу, на который указывает dest. Затем оба указателя увеличиваются и на следующей итерации передается следующий символ. Цикл прекращается, когда в src будет найден символ конца строки; в этой точке в dest присваивается значение null, работа функции завершена. На рис. 10.11 показано, как указатели движутся по строкам.

Библиотека строковых функций

Многие из использованных нами библиотечных функций для строк имеют строковые аргументы, которые определены с использованием указателей. В качестве примера вы можете посмотреть описание функции strcpy() в документации к

компилятору (или в заголовочном файле string.h). Эта функция предназначена для копирования одной строки в другую; вы можете сравнить ее с функцией copystr() из нашего примера COPYSTR. Рассмотрим синтаксис библиотечной функции srtcpy():

```
char* strcpy ( char* dest, const char* src );
```

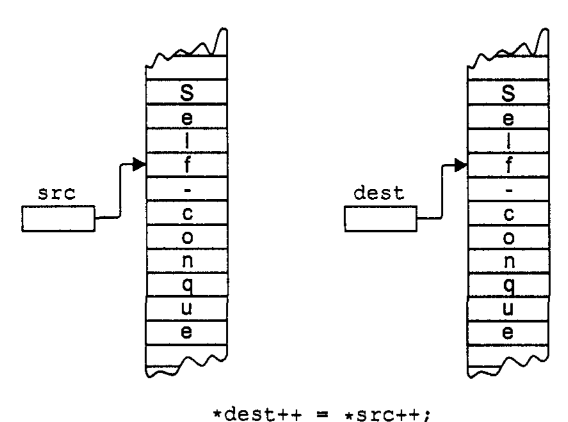


Рис. 10.11. Действия программы COPYSTR

Эта функция имеет два аргумента типа char*. (В следующем разделе «Модификатор const и указатели» мы объясним значение слова const в этом контексте.) Функция strcpy() возвращает указатель на char; это адрес строки dest. В других отношениях эта функция работает так же, как и написанная нами функция copystr().

Модификатор const и указатели

Использование модификатора const при объявлении указателя может сбивать с толку, потому что это может означать один из двух вариантов в зависимости от его расположения. В приведенных ниже строках описаны оба варианта:

```
const int* cptrInt; // указатель на константу
int* const ptrcInt; // константный указатель
```

Используя первый вариант объявления указателя, вы не можете изменять значение переменной, на которую указывает указатель cptrInt, но можете изменять значение самого указателя cptrInt. Если вы воспользуетесь вторым вариантом, то все будет наоборот. Вы не сможете изменять значение самого указателя cptrInt, но сможете изменять значение того, на что cptrInt указывает. Вы должны помнить различия в названиях этих указателей, которые указаны в комментариях. Можно использовать const в обеих позициях и сделать константами как сам указатель, так и то, на что он указывает.

В объявлении функции strcpy() показано, что параметр const char* scr определен так, что функция не может изменять строку, на которую указывает scr.

Это не значит, что указатель scr не может быть изменен. Для того чтобы указатель стал константой, нужно при его объявлении указать char* const scr.

Массивы указателей на строки

Массив указателей — это то же, что и массив переменных типа int или float. Обычно он используется в качестве массива указателей на строки.

В примере STRARAY главы 7 был продемонстрирован массив строк char*. Как мы заметили, использование массива строк нерационально по причине того, что все подмассивы, содержащие строки, должны быть одной длины, и пространство будет простаивать, если строка короче, чем длина подмассива (см. рис. 7.10 в главе 7).

Давайте рассмотрим, как использование указателей решает эту проблему. Мы модифицируем пример STRARAY и создадим массив указателей на строки вместо массива строк. Листинг программы PTRTOSTR:

```
// ptrtostr.cpp
// массив указателей на строки
#include <iostream>
using namespace std;
const int DAYS = 7;

int main ( )
{
  // массив указателей на строки
  char* arrptrs [ DAYS ] = { "Понедельник", "Вторник", "Среда", "Четверг", "Пятница",
"Суббота", "Воскресенье" };

  for ( int j = 0; j < DAYS; j++ ) // покажем все строки
    cout << arrptrs [ j ] << endl;

  return 0;
}
```

Результат работы этой программы будет тем же, что и программы STRARAY:

```
Понедельник
Вторник
Среда
Четверг
Пятница
Суббота
Воскресенье
```

В том случае, если строки не являются частью массива, то C++ размещает их в памяти друг за другом, чтобы не было пустого пространства. Однако для поиска строк создается массив, содержащий указатели на них. Сами строки — это массивы элементов типа char, поэтому массив указателей на строки является массивом указателей на char. Это и означает определение массива arrptrs в программе PTRTOSTR. Теперь вспомним, что адресом строки является адрес ее первого элемента. Именно эти адреса хранит массив указателей на строки. На рис. 10.12 показано, как это выглядит.

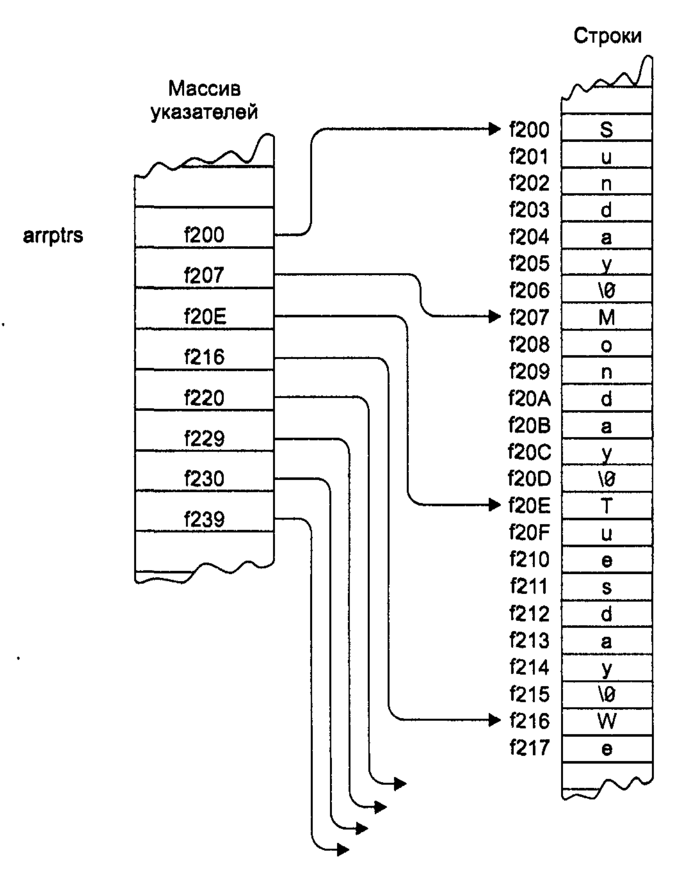


Рис. 10.12. Массив указателей и строки

Управление памятью: операции new и delete

Мы рассмотрели много примеров, в которых массивы использовались без учета *размера памяти. В строке*

```
int arr1 [ 100 ]:
```

зарезервирована память для 100 элементов типа int. Массивы являются разумным подходом к хранению данных, но они имеют серьезный недостаток: мы должны знать при написании программы, насколько большой массив нам нужен. Мы не можем ждать, пока программа запустится и определит размер массива. Следующий подход работать не будет:

```
cin >> size;      // получим желаемый размер массива
int arr [ size ]; // ошибка. размер массива должен быть константой!
```

Компилятор требует, чтобы значение размерности массива было константой.

Но во многих ситуациях нам неизвестно требуемое количество памяти до запуска программы. Например, нам нужно будет сохранить строку, которую напечатает пользователь программы. В этой ситуации мы можем определить размер массива большим, но это может привести к перерасходу памяти. (В главе 15 «Стандартная библиотека шаблонов (STL)» мы будем использовать вектор, являющийся видом расширяемого массива.)

Операция new

C++ предлагает другой подход к выделению памяти: операцию new. Это универсальная операция, получающая память у операционной системы и возвращающая указатель на начало выделенного блока. В программе NEWINTRO показано, как это работает:

```
// newintro.cpp
// познакомимся с оператором new
#include <iostream>
#include <cstring>
using namespace std;

int main ( )
{
  char* str = "Дурная голова ногам покоя не дает.";
  int len = strlen ( str );        // вычислим длину нашей строки

  char* ptr;                       // определим переменную
  ptr = new char [ len +1 ];       // выделим память

  strcpy ( ptr, str );             // скопируем строку str в ptr

  cout << "ptr = " << ptr << endl; // покажем что содержится в ptr

  delete [] ptr;                   // освободим выделенную память

  return 0;
}
```

Выражение

```
ptr = new char [ len + 1 ];
```

присваивает переменной ptr значение адреса блока памяти, достаточного для хранения строки str, длину которой можно получить, используя библиотечную функцию strlen() плюс дополнительный байт для символа конца строки. На рис. 10.13 показан синтаксис операции new. Обратите внимание на использование квадратных скобок. На рис. 10.14 показана память, выделенная с помощью операции new, и указатель на нее.

В примере NEWINTRO мы использовали функцию strcpy() для копирования строки str в выделенный участок памяти, на который указывает указатель ptr. Строка полностью займет этот участок памяти, так как его размер равен длине строки str. Результат работы программы NEWINTRO будет следующим:

```
ptr = Дурная голова ногам покоя не дает.
```

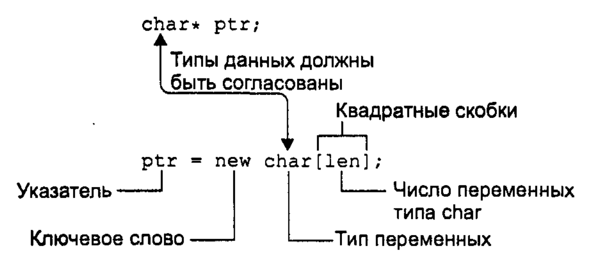


Рис. 10.13. Синтаксис операции new

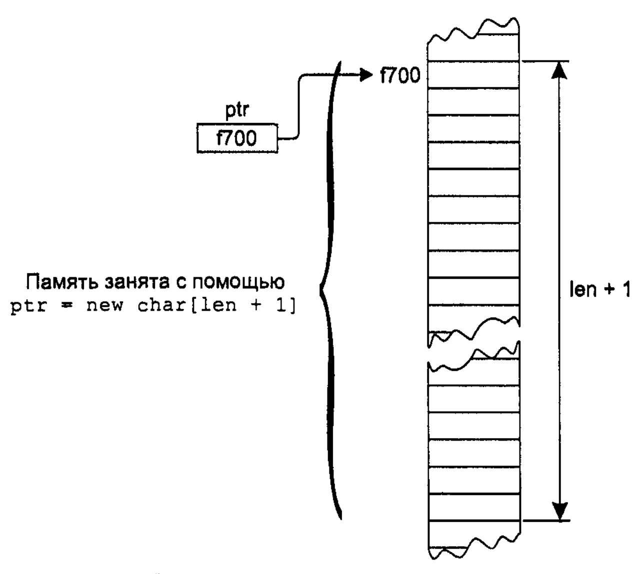


Рис. 10.14. Память, выделенная с помощью операции new

Программисты, использующие С, говорят, что оператор new играет роль, похожую на роль семейства функций malloc() библиотеки функций. Преимущество операции new в том, что он сразу возвращает указатель на соответствующий тип данных, в то время как указатель функции malloc() должен быть явно преобразован к соответствующему типу.

Программистам, использующим С, может быть интересно, существует ли в С++ функция для изменения размера памяти, которая была уже перераспределена, эквивалентная функции realloc(). К сожалению, операции renew в языке С++ не существует. Вам придется пойти на хитрость, выделить с помощью операции new большее или меньшее количество памяти, а затем скопировать ваши данные в новый блок памяти.

Операция delete

Если ваша программа резервирует много участков памяти, используя операцию new, то в конце концов все возможное пространство памяти будет занято и система рухнет. Для того чтобы избежать этого и эффективно использовать память,

используется операция delete. Она предназначена для освобождения выделенных участков памяти, возвращая их операционной системе. В примере NEWINTRO строка

```
delete [ ] ptr:
```

возвращает системе память, на которую указывал указатель ptr.

На самом деле в последнем примере нам не нужна эта операция, так как память автоматически освобождается после завершения работы программы. Однако предположим, что мы используем операцию new в функции. Если функция использует локальную переменную как указатель на память, выделенную с использованием new, то указатель будет уничтожен по окончании работы функции, но память останется выделенной и при этом недоступной из программы. Поэтому освобождение памяти после того, как она была использована и больше не требуется, является хорошим тоном, а зачастую и просто необходимо.

Освобождение памяти не подразумевает удаление указателя, связанного с этим блоком памяти; это и не изменение адреса значения, на которое указывает указатель. Однако этот указатель не имеет силы; память, на которую он указывает, может быть применена для других целей. Поэтому будьте внимательны и не используйте указатели на освобожденную память.

Квадратные скобки, следующие за операцией delete, означают, что мы освобождаем массив. При освобождении памяти, выделенной для одиночного объекта, использования скобок не требуется.

```
ptr = new SomeClass: // создаем один объект
...
delete ptr;          // скобок не требуется
```

Однако при освобождении памяти, занимаемой массивом, скобки обязательны. Их использование подразумевает, что мы освобождаем память, используемую для всех членов массива, одновременно.

Класс String с использованием операции new

Операция new часто используется в конструкторах классов. Для примера мы модифицируем класс String, который рассматривали в примере STRPLUS главы 8 «Перегрузка операций». Мы можем устранить возможный дефект этого класса, возникающий, когда все объекты класса String занимают одинаковое количество памяти. Строки, занимающие меньший объем памяти, расходуют память впустую, а строки, которые ошибочно сгенерированы больше, чем предоставленный им объем, могут привести к ошибке программы, так как они выходят за пределы массива. В нашем следующем примере мы используем операцию new для выделения ровно необходимого количества памяти. Вот листинг программы NEWSTR:

```
// newstr.cpp
// использование оператора new для выделения памяти под строку
#include <iostream>
#include <cstring>
using namespace std:
//////////////////////////////////////////////////////////////
```

```
class String
{
  private:
    char* str;          // указатель на строку
  public:
    String ( char* s )  // конструктор с одним параметром
      {
        int length = strlen ( s );     // вычисляем длину строки
        str = new char [ length + 1 ]; // выделяем необходимую память
        strcpy ( str, s );             // копируем строку
      }
    ~String ( )         // деструктор
      {
        cout << "Удаляем строку\n";
        delete [ ] str; // освобождаем память
      }
    void display ( )    // покажем строку на экране
      {
        cout << str << endl;
      }
}
///////////////////////////////////////////////////////////
int main ( )
{
  String s1 = "Тише едешь - дальше будешь.";
  cout << "s1 = ";
  s1.display ( );

  return 0;
}
```

Результат работы программы будет следующим:

```
s1 = Тише едешь - дальше будешь.
Удаляем строку
```

Класс String имеет только одно поле str, являющееся указателем на char. Он будет указывать на строку, содержащуюся в объекте класса String. Здесь не будет массива, содержащего строку. Строка хранится в другом месте, а в классе String есть только указатель на нее.

Конструктор в программе NEWSTR

Конструктор в этом примере имеет один аргумент: указатель на строку char*. С его помощью, используя операцию new, выделяется память для строки, на которую будет указывать указатель str. Затем конструктор использует функцию strcpy() для копирования строки в выделенный участок памяти.

Деструктор в программе NEWSTR

Мы нечасто использовали деструктор в наших примерах, но теперь он нам необходим для освобождения памяти, выделенной с помощью операции new. При создании объекта мы выделяем для него память, и будет вполне разумным освободить эту память после того, как объект уже использован. Как вы могли заметить в главе 6, деструктор — это функция, которая вызывается автоматически

при удалении объекта. В программе NEWSTR деструктор выглядит следующим образом:

```
~String()
{
  cout << "Удаляем строку.\n";
  delete [ ] str;
}
```

Деструктор возвращает системе память, выделенную при создании объекта. Мы можем вывести в конце программы сообщение о том, что деструктор выполнил свою функцию. Объекты (как и другие переменные) обычно уничтожаются, когда функция, в которой они определены, заканчивает свою работу. Деструктор гарантирует, что память, выделенная для объекта класса String, будет возвращена системе при уничтожении объекта, а не останется неопределенной.

Мы должны заметить, что при использовании деструктора, как это показано в примере NEWSTR, могут возникнуть некоторые проблемы. При копировании одного объекта класса String в другой в строке s1=s2 в действительности происходит копирование указателя на реальную строку. Теперь оба объекта указывают на одну и ту же строку. При удалении строки деструктор удалит один указатель на строку, а указатель другого объекта будет указывать уже на что-то другое. Мы можем этого не заметить, так как процесс удаления объектов для нас невидим, например когда заканчивает свою работу функция, в которой был создан локальный объект. В главе 11 мы рассмотрим, как создать деструктор, который подсчитывает, сколько объектов класса String указывают на строки.

Указатели на объекты

Указатели могут указывать на объекты так же, как и на простые типы данных и массивы. Мы рассмотрели много примеров объявления таких объектов, как:

```
Distance dist;
```

где определен объект dist класса Distance.

Однако в некоторых случаях на момент написания программы нам неизвестно количество объектов, которое необходимо создать. Тогда мы можем использовать операцию new для создания объектов во время работы программы. Как мы видели, операция new возвращает указатель на неименованный объект. Рассмотрим два подхода к созданию объектов на примере небольшой программы ENGLPTR.

```
// englptr.cpp
// доступ к членам класса через указатель
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance
{
  private:
    int feet;
    float inches;
```

```
  public:
    void getdist ( ) // получение информации
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: "; cin >> inches;
      }
    void showdist ( ) // вывод информации
      { cout << feet << "\'-" << inches << '\"'; }
}
//////////////////////////////////////////////////////////
int  main ( )
{
  Distance dist;     // определяем объект типа Distance
  dist.getdist ( );  // получаем информацию
  dist.showdist ( ); // выводим на экран

  Distance* distptr;       // определяем переменную-указатель на Distance
  distptr = new Distance;  // создаем объект Distance
  distptr->getdist ( );    // получаем для него информацию
  distptr->showdist ( );   // выводим информацию
  cout << endl;

  return 0;
}
```

В этой программе использована английская вариация класса Distance, рассмотренная в предыдущей главе. В функции main() определен dist, использующий сначала функцию getdist() класса Distance для получения интервала от пользователя, а затем функцию showdist() для вывода на дисплей этого интервала.

Ссылки на члены класса

В программе ENGLPTR затем создается другой объект типа Distance с использованием операции new, которая возвращает указатель на него, именуемый distptr.

Возникает вопрос: как мы ссылаемся на методы класса из объекта, на который указывает distptr? Вы можете догадаться, что в этом случае мы можем использовать точку (.), операцию доступа к членам класса:

```
distptr.getdist ( ); // так нельзя: distptr не просто переменная
```

но это не будет работать. Операция точки требует, чтобы идентификатор слева был переменной. Так как distptr — указатель на переменную, то в этом случае применяется другой синтаксис. Это другой случай использования операции *разыменования* (операции получения содержимого переменной, на которую указывает указатель):

```
(*distptr).getdist ( ); // так можно: разыменование указателя это переменная
```

Однако это немного неуклюже из-за скобок (скобки необходимы, так как операция точки (.) имеет больший приоритет, чем операция разыменования (*)). Эквивалентный, но более краткий способ записи предоставляется операцией доступа к членам класса, он состоит из дефиса и знака *больше чем*:

```
distptr->getdist ( ); // лучший вариант
```

Как вы могли увидеть в примере ENGLPTR, операция -> работает с указателями на объекты так же, как и операция точки работает с объектами. Рассмотрим результат работы программы:

```
Введите футы: 10
Введите дюймы: 6.25
10'-6.25"

Введите футы: 6
Введите дюймы: 4.75
6'-4.75"
```

Другое применение операции new

Вам может встретиться другой, более общий способ применения операции new для выделения памяти для объектов. Так как new возвращает указатель на блок памяти, содержащий объект, то мы имеем возможность обратиться к первоначальному объекту путем разыменования указателя. В примере ENGLREF показано, как это можно сделать:

```
// englref.cpp
// разыменование указателя, возвращаемого оператором new
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance
{
  private:
    int feet;
    float inches;
  public:
    void getdist ( ) // получение информации
      {
        cout << "\nВведите футы: "; cin >> feet;
        cout << "Введите дюймы: "; cin >> inches;
      }
    void showdist ( ) // вывод информации
      { cout << feet << "\'-" << inches << '\"'; }
};
///////////////////////////////////////////////////////////
int main ( )
{
  Distance& dist = *( new Distance ); // создаем объект типа Distance
  dist.getdist ( );                   // доступ к членам класса осуществляем через оператор
"."
  dist.showdist ( );
  cout << endl;

  return 0;
}
```

Выражение

```
new Distance;
```

возвращает указатель на блок памяти, достаточный для размещения объекта Distance, и мы можем обратиться к первоначальному объекту, используя

```
*( new Distance ):
```

На этот объект указывает указатель. Используя ссылку, мы определили dist как объект класса Distance и определили его равным *(new Distance). Теперь мы можем ссылаться на члены объекта dist, используя операцию точки, а не операцию ->.

Этот подход применяется реже, чем указатели на объекты, полученные с использованием операции new, или простое объявление объекта, но он работает точно так же.

Массив указателей на объекты

Массив указателей на объекты — это часто встречающаяся конструкция в программировании. Этот механизм упрощает доступ к группе объектов, он более гибок, чем просто создание массива объектов. (Например, в программе PERSORT этой главы мы увидим, как группа объектов может быть отсортирована путем использования сортировки массива указателей на них.)

В нашем следующем примере PTROBJS мы создадим массив указателей на объекты класса person. Вот листинг программы:

```
// ptrobjs.cpp
// массив указателей на объекты
#include <iostream>
using namespace std:
///////////////////////////////////////////////////////////
class person // класс человек
{
  private:
    char name [ 40 ]:      // имя человека
  public:
    void setName ( )       // установка имени
      {
        cout << "Введите имя: ":
        cin >> name:
      }
    void printName ( )     // показ имени
      {
        cout << "\n  Имя: " << name:
      }
}:
///////////////////////////////////////////////////////////
int main ( )
{
  person* persPtr [ 100 ]: // массив указателей
  int n = 0:               // количество элементов в массиве
  char choice:

  do
  {
    persPtr [ n ] = new person:       // создаем новый объект
    persPtr [ n ]->setName ( ):       // вводим имя
```

```
    n++;                              // увеличиваем количество
    cout << "Продолжаем ввод (д/н)?"; // спрашиваем, закончен ли ввод
    cin >> choice;
  }
  while ( choice == 'д' );
  for ( int j = 0; j < n; j++ )
  {
    cout << "\nИнформация о номере " << j + 1;
    persPtr [ j ]->printName ( );
  }
  cout << endl;

  return 0;
}
```

Класс person имеет одно поле name, которое содержит строку с именем клиента. Методы класса setName() и printName() позволяют нам ввести имя и затем вывести его на дисплей.

Действия программы

В функции main() определен массив persPtr, содержащий 100 указателей на объекты типа person. Мы вводим имя клиента, запрашивая пользователя в цикле. Для каждого имени с использованием операции new создается объект класса person, указатель на который хранится в массиве persPtr. Продемонстрируем, как можно легко получить доступ к объектам, используя указатели, а затем распечатать для каждого объекта person поле name.

Это пример работы с программой:

```
Введите имя: Иванов
Продолжаем ввод (д/н)? д
Введите имя: Петренко
Продолжаем ввод (д/н)? д
Введите имя: Абдурахманов
Продолжаем ввод (д/н)? н
Информация о номере 1
  Имя: Иванов
Информация о номере 2
  Имя: Петренко
Информация о номере 3
  Имя: Абдурахманов
```

Доступ к методам класса

Для объектов класса person, на которые указывают указатели массива persPtr, нам нужен доступ к методам класса person setName() и printName(). Каждый из элементов массива persPtr определен в цикле с использованием операций массива persPtr[j] (при использовании указателей это будет выглядеть как *(persPtr+j)). Элементами массива являются указатели на объекты типа person. Для доступа к членам объектов с использованием указателя применим операцию ->. С учетом сказанного выше получим следующий синтаксис для обращения к методу getName():

```
persPtr [ j ]->getName ( );
```

Таким образом, мы вызываем метод getName() для объекта класса person, на который указывает указатель, являющийся j элементом массива persPtr.

Связный список

В нашем следующем примере мы рассмотрим простой связный список. Что это такое? Связный список тоже предназначен для хранения данных. До сих пор мы рассматривали как пример структур для хранения данных лишь массивы и массивы указателей на данные (в примерах PTRTOSTRS и PTROBJS). Недостатком обеих этих структур является необходимость объявлять фиксированное значение размера массива до запуска программы.

Цепочка указателей

Связный список представляет собой более гибкую систему для хранения данных, в которой вовсе не используются массивы. Вместо них мы получаем для каждого элемента память, используя операцию new, а затем соединяем (или *связываем*) элементы данных между собой, используя указатели. Элементы списка не обязаны храниться в смежных участках памяти, как это происходит с массивами; они могут быть разбросаны по всему пространству памяти.

В нашем примере связный список является объектом класса linkList. Элементы списка представлены структурой link. Каждый элемент содержит целое число, представляющее собой данные, и указатель на следующий элемент списка. Указатель на сам список хранится в начале списка. Структура списка показана на рис. 10.15.

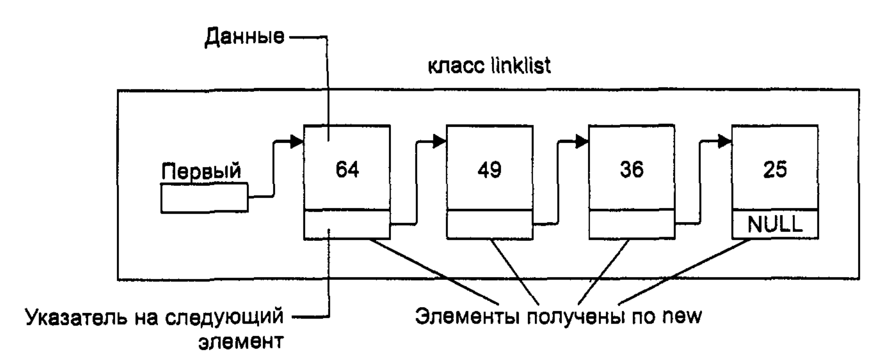


Рис. 10.15. Связный список

Рассмотрим листинг программы LINKLIST:

```
// linklist.cpp
// список
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
```

```
struct link   // один элемент списка
{
  int data;   // некоторые данные
  link* next; // указатель на следующую структуру
}
//////////////////////////////////////////////////////////
class linklist // список
{
  private:
    link* first;
  public:
    linklist ( )             // конструктор без параметров
      { first = NULL; }      // первого элемента пока нет
    void additem ( int d ); // добавление элемента
    void display ( );        // показ данных
};
//////////////////////////////////////////////////////////
void linklist::additem ( int d ) // добавление элемента
{
  link* newlink = new link;      // выделяем память
  newlink->data = d;             // запоминаем данные
  newlink->next = first;         // запоминаем значение first
  first = newlink;               // first теперь указывает на новый элемент
}
//////////////////////////////////////////////////////////
void linklist::display ( )
{
  link* current = first;           // начинаем с первого элемента
  while( current )                 // пока есть данные
  {
    cout << current->data << endl; // печатаем данные
    current = current->next;       // двигаемся к следующему элементу
  }
}
//////////////////////////////////////////////////////////
int main ( )
{
  linklist li;       // создаем переменную-список

  li.additem ( 25 ); // добавляем туда несколько чисел
  li.additem ( 36 );
  li.additem ( 49 );
  li.additem ( 64 );

  li.display ( );    // показываем список

  return 0;
}
```

Класс linklist имеет только одно поле: указатель на начало списка. При создании списка конструктор инициализирует этот указатель, именованный как first, значением NULL, которое аналогично значению 0. Это значение является признаком того, что указатель указывает на адрес, который точно не может содержать полезной информации. В нашей программе элемент, указатель которого на следующий элемент имеет значение NULL, является конечным элементом списка.

Добавление новых элементов в список

Функция additem() позволяет нам добавить новый элемент в связный список. Новый элемент вставляется в начало списка (мы могли бы вставлять новый элемент и в конец списка, но это будет более сложным примером). Давайте последовательно рассмотрим действия при вставке нового элемента.

Сначала мы создаем новый элемент типа link в строке

```
link* newlink = new link;
```

Таким образом, мы выделяем память для нового элемента с помощью операции new и сохраняем указатель на него в переменной newlink.

Затем мы заполняем элемент нужным нам значением. При этом действия со структурой похожи на действия с классами; к элементам структуры можно получить доступ, используя операцию ->. В следующих двух строках программы мы присваиваем переменной data значение, переданное как аргумент функции additem(), а указателю на следующий элемент мы присваиваем значение, хранящееся в указателе first. Этот указатель содержит в себе адрес начала списка.

```
newlink->data = d;
newlink->next = first;
```

И в заключение мы присваиваем указателю first значение указателя на новый элемент списка.

```
first = newlink;
```

Смыслом всех этих действий является замена адреса, содержащегося в указателе first, на адрес нового элемента, при этом старый адрес указателя first превратится в адрес второго элемента списка. Этот процесс иллюстрирует рис. 10.16.

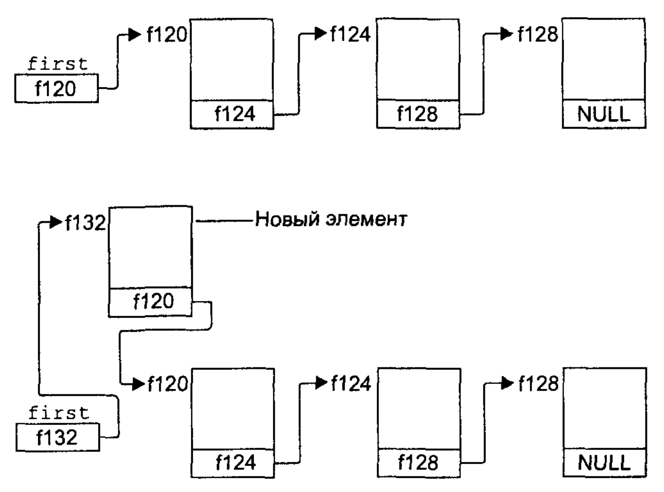


Рис. 10.16. Вставка нового элемента в связный список

Получение содержимого списка

Когда список уже создан, легко получить хранящиеся в нем значения (или выполнить операции с ними). Все, что нам нужно, это следовать от одного указателя next к другому до тех пор, пока его значение не станет равным NULL, означающим конец списка. В строке функции display()

```
cout << endl << current->data;
```

выводится значение, содержащееся в переменной data, и

```
current = current->next;
```

перемещает нас к следующему элементу списка до тех пор, пока

```
current != NULL;
```

выражение условия в операторе while не станет ложным. Вот результат работы программы LINKLIST:

```
64
49
36
25
```

Связные списки — это наиболее часто встречающаяся после массивов структура для хранения данных. Как мы заметили, в них, в отличие от массивов, устранена возможная потеря памяти. Недостаток списков в том, что для поиска определенного элемента списка необходимо пройти сквозь весь список от его начала до нужного элемента. И это может занять много времени. А к элементам массива мы можем получить мгновенный доступ, используя индекс элемента, известный заранее. Более подробно мы рассмотрим связные списки и другие структуры для хранения данных в главе 15 «Стандартная библиотека шаблонов (STL)».

Классы, содержащие сами себя

Рассмотрим возможные ошибки при использовании классов и структур, ссылающихся на самих себя. Структура link в примере LINKLIST содержит указатель на такую структуру. Вы можете проделать то же и с классами:

```
class sampleclass
{
  sampleclass* ptr; // так можно
};
```

Однако хотя класс может содержать в себе указатель на такой же объект, сам этот объект он содержать в себе не может:

```
class sampleclass
{
  sampleclass obj; // так нельзя
};
```

Это справедливо как для классов, так и для структур.

Пополнение примера LINKLIST

Основа примера LINKLIST может быть использована не только для рассмотренного случая, но и для более сложных примеров, когда каждый элемент содержит больше данных, а вместо целого числа — порядковый номер элемента или указатель на структуру или объект.

Новые функции могут выполнять такие действия, как добавление или удаление элемента из любой части списка. Еще одна важная функция — это деструктор. Как было замечено раньше, важно освободить неиспользуемые блоки памяти. Поэтому деструктор, предназначенный для этой операции, необходимо добавить в класс linklist. Используя в нем операцию delete, мы сможем освободить выделенные для каждого элемента списка участки памяти.

Указатели на указатели

В нашем следующем примере мы рассмотрим массив указателей на объекты и покажем, как можно отсортировать эти указатели, основываясь на данных, содержащихся в объектах. Здесь появляется идея использования указателей на указатели. Мы также продемонстрируем, почему люди иногда теряют сон из-за указателей.

Задача следующей программы: создать массив указателей на объекты класса person. Она будет похожа на пример PTROBJS, но мы еще добавим варианты функций order() и bsort() из примера PTRSORT для сортировки группы объектов класса person, основанной на алфавитном порядке имен. Вот листинг программы PERSORT:

```
// persort.cpp
// сортировка объектов через массив указателей на них
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class person            // некоторый человек
{
  protected:
    string name;        // имя
  public:
    void setName ( )    // установка имени
      { cout << "Введите имя: " cin >> name; }
    void printName ( )  // показ имени
      { cout << endl << name; }
    string getName ( )  // получение имени
      { return name; }
};
///////////////////////////////////////////////////////////
int main ( )
{
  void bsort ( person**, int ); // прототип функции
  person* persPtr [ 100 ];      // массив указателей на person
  int n = 0;                    // количество элементов в массиве
  char choice;                  // переменная для ввода символа

  do
  {
```

```
    persPtr [ n ] = new person:       // создаем новый объект
    persPtr [ n ]->setName ( ):       // вводим имя
    n++:                              // увеличиваем количество
    cout << "Продолжаем ввод (д/н)?": // спрашиваем, закончен ли ввод
    cin >> choice:
  }
  while ( choice == 'д' ):

  cout << "\nНеотсортированный список:":
  for ( int j = 0: j < n: j++ ) // покажем неотсортированный список
    persPtr [ j ]->printName ( ):

  bsort ( persPtr, n ):         // отсортируем указатели

  cout << "\nОтсортированный список:":
  for ( j = 0: j < n: j++ )     // покажем отсортированный список
    persPtr [ j ]->printName ( ):
  cout << endl:

  return 0:
}
///////////////////////////////////////////////////////////////
void bsort ( person** pp, int n )
{
  void order ( person**, person** ): // прототип функции
  int j, k:                          // переменные для циклов

  for ( j = 0: j < n - 1: j++ )      // внешний цикл
    for ( k = j + 1: k < n: k++ )    // внутренний цикл
      order ( pp + j, pp + k ):      // сортируем два элемента
}
///////////////////////////////////////////////////////////////
void order ( person** pp1, person** pp2 )
{
  if( (*pp1)->getName ( ) > (*pp2)->getName ( ) ) // если первая строка больше второй,
  {
    person* tempptr = *pp1: // меняем их местами
    *pp1 = *pp2:
    *pp2 = tempptr:
  }
}
```

После запуска программы запрашивается имя. Когда пользователь введет его, создается объект типа person и полю data присваивается имя, полученное от пользователя. В программе также хранятся в виде массива persPtr указатели на эти объекты.

Когда пользователь на запрос о продолжении ввода имен вводит букву «н», программа вызывает функцию bsort() для сортировки объектов типа person, основанной на содержимом полей name. Вот небольшой пример работы с программой:

```
Введите имя: Иванов
Продолжаем ввод (д/н)? д
Введите имя: Петренко
Продолжаем ввод (д/н)? д
Введите имя: Абдурахманов
Продолжаем ввод (д/н)? н
Неотсортированный список:
```

```
Иванов
Петренко
Абдурахманов

Отсортированный список:
Абдурахманов
Иванов
Петренко
```

Сортируем указатели

В действительности, когда мы сортируем объекты типа person, мы не трогаем сами объекты, а работаем с указателями на них. Таким образом мы исключаем необходимость перемещения объектов в памяти, которое отнимает много времени, если объекты очень большие. Это также позволяет нам выполнять сложные сортировки одновременно — например, одну по имени, а другую по телефонным номерам — без многократного сохранения объектов. Этот процесс показан на рис. 10.17.

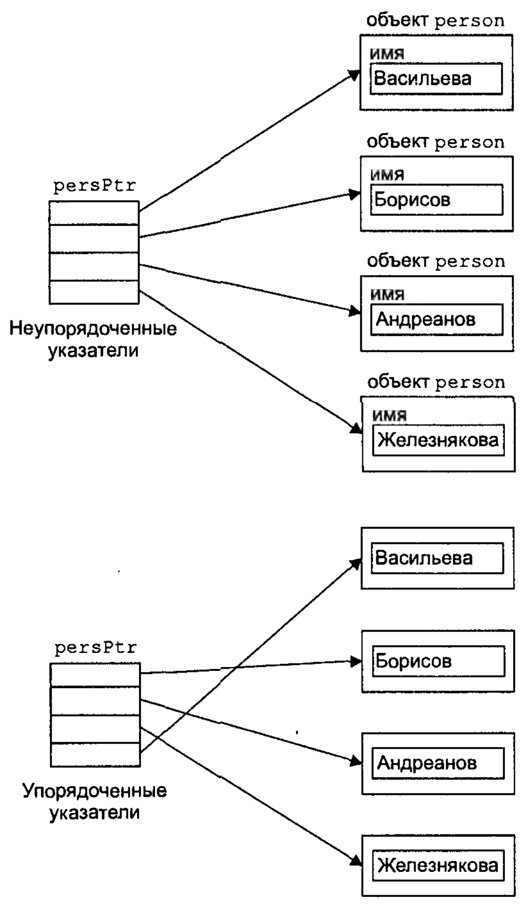


Рис. 10.17. Сортировка массива указателей

Для облегчения сортировки мы добавили в класс person метод getName(), дающий доступ к именам из функции order() и решающий, где менять указатели.

Тип данных person**

Наверное, вы заметили, что первый аргумент функции bsort() и оба аргумента функции order() имеют тип person**. Что означают эти две звездочки? Эти аргументы используются для передачи адреса массива persPtr или, в случае функции order(), адресов элементов массива. Если это массив типа person, то адрес массива будет типа person*. Однако массив *указателей* на person имеет адреса своих элементов типа person**. Адрес указателя — это указатель на указатель. На рис. 10.18 показано, как это выглядит.

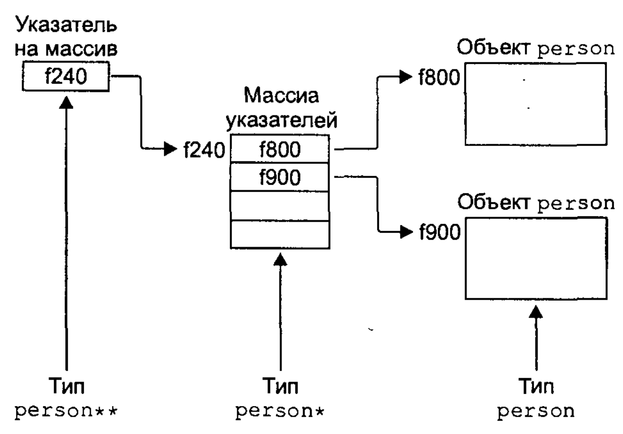


Рис. 10.18. Указатель на массив указателей

Сравним эту программу с примером PTRSORT, в котором сортируем массив типа int. Мы увидим, что тип данных, передающихся функции в программе PERSORT, имеет еще одну звездочку в отличие от типа данных в примере PTRSORT, так как массив является массивом указателей.

Так как массив persPtr содержит указатели, то конструкция

```
persPtr [ j ]->printName ( );
```

выполняет метод printName() для объекта, на который указывает j элемент массива persPtr.

Сравнение строк

В программе PERSORT функция order() была модифицирована для лексикографического сравнения двух строк, то есть для их расположения в алфавитном порядке. Для проведения этого сравнения мы воспользовались библиотечной функцией C++ strcmp(). Эта функция принимает две строки s1 и s2 в качестве аргументов strcmp(s1, s2) и возвращает одно из следующих значений:

Значение	Описание
< 0	s1 идет перед s2
0	s1 и s2 одинаковы
> 0	s1 идет после s2

Доступ к строке мы получаем, используя следующий синтаксис:

```
(*pp1)->getName ( );
```

Аргумент pp1 является указателем на указатель, а мы хотим получить содержимое переменной, находящейся по адресу, на который указывает указатель. Для разыменования одного указателя используется операция ->, а для разыменования второго указателя используется операция звездочка, предшествующая указателю pp1.

Так как существуют указатели на указатели, то могут быть и указатели на указатели, которые указывают на указатели и т. д. К счастью, такие сложные конструкции редко встречаются.

Пример разбора строки

Программисты часто встречаются с проблемой осуществления *разбора* строки символов. Примерами строк могут являться команды, вводимые пользователем с клавиатуры, предложения естественных языков (таких, как русский), операторы языков программирования и алгебраические выражения. При изучении указателей и строк мы можем встретиться с этой проблемой.

В нашем следующем примере мы покажем, как разобрать арифметическое выражение типа:

```
6/3+2*3-1
```

Пользователь вводит выражение, программа обрабатывает его знак за знаком, определяя, что означает каждый знак в арифметических терминах, и выдает результат (7 для нашего примера). В этом выражении использованы четыре арифметических операции: +, -, * и /. Мы упростили нашу задачу, ограничившись числами от 0 до 9. Также мы не будем рассматривать скобки.

В этой программе мы будем использовать класс Stack (см. пример STAKARAY главы 7). Модифицируем этот класс, чтобы он мог хранить данные типа char. Мы используем стек для хранения выражения в виде символов. Стек удобно использовать при разборе выражений. Нам часто приходится обращаться к последнему из сохраненных символов, а стек является контейнером типа «последний вошел — первый вышел».

Кроме класса Stack, мы будем использовать класс express, представляющий целые арифметические выражения. Методы этого класса позволят нам инициализировать объект выражением в форме строки (введенной пользователем), разобрать это выражение и возвратить результат арифметического выражения.

Разбор арифметических выражений

Этот раздел посвящен разбору арифметических выражений. Начнем слева и рассмотрим все символы по очереди. Это могут быть *числа* (от 0 до 9) или *операции* (знаки +, -, * и /).

Если символ представляет собой число, то мы помещаем его в стек. Туда же мы поместим первую из встретившихся нам операций. Вся хитрость заключается в том, как составить последовательность операций. Заметим, что мы не выполняем действие текущей операции, так как мы еще не знаем, какое за ней следует число. Найденная операция является сигналом, что мы можем выполнить предыдущую операцию, которая помещена в стек. Так, если в стеке последовательность 2 + 3, то перед выполнением сложения мы должны найти следующую операцию.

Таким образом, когда мы поняли, что текущий символ является операцией (за исключением первого), мы извлекаем из стека предыдущее число (3 в нашем примере) и предыдущую операцию (+) и помещаем их в переменные lastval и lastop. Наконец мы извлекаем первое число (2) и выполняем арифметическую операцию с двумя числами (получая 5). Всегда ли мы можем выполнить предыдущую операцию? Нет. Вспомним, что * и / имеют более высокий приоритет, чем + и -. В выражении 3+4/2 мы не можем выполнить сложение пока не произведено деление. Поэтому, когда мы встретим операцию / в этом выражении, то мы должны положить 2 и + обратно в стек до тех пор, пока не будет выполнено деление.

С другой стороны, если текущая операция + или -, то мы всегда можем выполнить предыдущую операцию. Так, когда мы встретим + в выражении 4–5+6, то мы можем выполнить операцию -, а когда мы увидим - в выражении 6/2–3, то мы можем выполнить деление. В таблице 10.1 отражены четыре возможных случая.

Таблица 10.1. Операции при разборе выражений

Предыдущий оператор	Текущий оператор	Пример	Действие1
+ или -	* или /1	3+4/	Положить в стек предыдущий оператор и предыдущее число (+, 4)
* или /	* или /	9/3*	Выполнить предыдущий оператор, положить в стек результат (3)
+ или -	+ или -	6+3+	Выполнить предыдущий оператор, положить в стек результат (9)
* или /	+ или -	8/2-	Выполнить предыдущий оператор, положить в стек результат (4)

Метод parse() выполняет этот процесс, последовательно обрабатывая полученное выражение и выполняя операции. Но здесь много работы. В стеке все еще содержится или одно число, или несколько последовательностей число—операция—число. Обрабатывая содержимое стека, мы выполняем операции этих последовательностей. Наконец в стеке останется одно число; оно будет значением первоначального выражения. Метод solve() выполняет все выше перечисленные

действия до тех пор, пока в стеке не останется одно число. Проще говоря, метод parse ( ) помещает элементы выражения в стек, а метод solve ( ) извлекает их.

Программа PARSE

Типичные действия с программой PARSE могут выглядеть следующим образом:

```
Введите арифметическое выражение в виде 2+3*4/3-2
Числа должны быть из одной цифры
Не используйте пробелы и скобки
Выражение: 9+6/3

Результат: 11
Еще одно выражение (д/н)? н
```

Заметим, что *результат* может состоять более чем из одной цифры. Он ограничен только численным размером типа char, от -128 до +127. Мы ограничили лишь вводимые числа диапазоном от 0 до 9.

Вот листинг программы:

```
// parse.cpp
// программа разбора арифметических выражений
#include <iostream>
#include <cstring>
using namespace std;
const int LEN = 80;         // максимальная длина выражения
const int MAX = 40;         //
//////////////////////////////////////////////////////////
class Stack
{
  private:
    char st [ MAX ];        // массив данных
    int top;                // количество сохраненных данных
  public:
    Stack ( )                // конструктор
      { top = 0; }
    void push ( char var ) // поместить в стек
      { st [ ++top ] = var; }
    char pop ( )             // взять из стека
      { return st [ top-- ]; }
    int gettop ( )           // узнать количество элементов
      { return top; }
};
//////////////////////////////////////////////////////////
class express
{
  private:
    Stack s;                    // стек данных
    char* pStr;                 // строка для ввода
    int len;                    // длина строки
  public:
    express ( char* ptr )       // конструктор
      {
        pStr = ptr;             // запоминаем указатель на строку
        len = strlen ( pStr ); // устанавливаем длину
      }
```

```
    void parse ( );            // разбор выражения
    int solve ( );             // получение результата
}
///////////////////////////////////////////////////////////
void express::parse ( ) // добавляем данные в стек
{
  char ch;             // символ из строки
  char lastval;        // последнее значение
  char lastop;         // последний оператор

  for ( int j = 0; j < len; j++ ) // для всех символов в строке
  {
    ch = pStr [ j ];   // получаем символ

    if ( ch >= '0' && ch <= '9' ) // если это цифра,
      s.push ( ch - '0' );        // то сохраняем ее значение
    else
    if ( ch == '+' || ch == '-' || ch == '*' || ch == '/' )
    {
      if ( s.gettop ( ) == 1 ) // если это первый оператор,
        s.push ( ch );         // помещаем его в стек
      else // иначе
      {
        lastval = s.pop ( ); // получаем предыдущее число
        lastop = s.pop ( );  // получаем предыдущий оператор
        // если это * или /, а предыдущий был + или -, то
        if ( ( ch == '*' || ch == '/' ) && ( lastop == '+' || lastop == '
        {
          s.push ( lastop ); // отменяем последние два взятия из стека
          s.push ( lastval );
        }
        else
        {
          // помещаем в стек результат операции
          switch ( lastop )
          {
            case '+': s.push ( s.pop ( ) + lastval ); break;
            case '-': s.push ( s.pop ( ) - lastval ); break;
            case '*': s.push ( s.pop ( ) * lastval ); break;
            case '/': s.push ( s.pop ( ) / lastval ); break;
            default: cout << "\nНеизвестный оператор"; exit ( 1 );
          }
        }
        s.push ( ch ); // помещаем в стек текущий оператор
      }
    }
    else               // какая-то ерунда...
    {
      cout << "\nНеизвестный символ";
      exit ( 1 );
    }
  }
}
///////////////////////////////////////////////////////////
int express::solve ( ) // убираем данные из стека
{
  char lastval;        // предыдущее значение

  while ( s.gettop ( ) > 1 )
```

```
  {
    lastval = s.pop ( ); // получаем предыдущее значение
    switch ( s.pop ( ) ) // получаем предыдущий оператор
    {
      case '+': s.push ( s.pop ( ) + lastval ); break;
      case '-': s.push ( s.pop ( ) - lastval ); break;
      case '*': s.push ( s.pop ( ) * lastval ); break;
      case '/': s.push ( s.pop ( ) / lastval ); break;
      default: cout << "\nНеизвестный оператор"; exit ( 1 );
    }
  }
  return int ( s.pop ( ) ); // последний оператор в стеке - это результат
}
///////////////////////////////////////////////////////////
int main ( )
{
  char ans;           // 'д' или 'н'
  char string [ LEN ]; // строка для разбора

  cout << "\nВведите арифметическое выражение в виде 2+3*4/3-2"
          "\nЧисла должны быть из одной цифры"
          "\nНе используйте пробелы и скобки";

  do
  {
    cout << "\nВыражение: ";
    cin >> string;                               // вводим строку
    express* eptr = new express ( string );      // создаем объект для разбора
    eptr->parse ( );                             // разбираем
    cout << "\nРезультат: "
         << eptr->solve ( );                     // решаем
    delete eptr;                                 // удаляем объект
    cout << "Еще одно выражение (д/н)? ";
    cin >> ans;
  }
  while ( ans == 'д' );
  return 0;
}
```

Это длинная программа, но в ней показано, как разработанный ранее класс Stack был применен в другой ситуации. Кроме того, программа демонстрирует различные случаи использования указателей и показывает, как полезно представлять строки в виде массива символов.

Симулятор: лошадиные скачки

В качестве последнего примера этой главы мы разберем игру «Лошадиные скачки». В этой игре номер лошади указан на экране. Лошадь стартует слева и скачет до финишной линии справа. В этой программе мы немного коснемся ООП и рассмотрим еще одну ситуацию с применением указателей.

Скорость каждой лошади выбрана случайным образом, поэтому невозможно вычислить заранее, какая из них выиграет. Программа использует консольную графику, и лошади легко, хотя и немного грубо, отражаются на дисплее. Нам

нужно будет скомпилировать эту программу вместе с заголовочными файлами MSOFTCON.H или BORLACON.H (это зависит от вашего компилятора) и файлами MSOFTCON.CPP или BORLACON.CPP (смотрите приложение Д «Упрощенный вариант консольной графики»).

При запуске программа HORSE запрашивает у пользователя данные о дистанции скачек и количестве лошадей, которые принимают в них участие. Классическим участком дистанции в лошадиных скачках (по крайней мере, в англоязычных странах) является 1/8 мили. Обычно дистанция включает в себя 6, 8, 10 или 12 таких участков. Бежать могут от 1 до 7 лошадей. Программа отображает границы каждого участка скачки, а так же линии старта и финиша, вертикальными линиями. Каждая лошадь представляет собой прямоугольник с номером посередине. На рис. 10.19 показана работа программы.

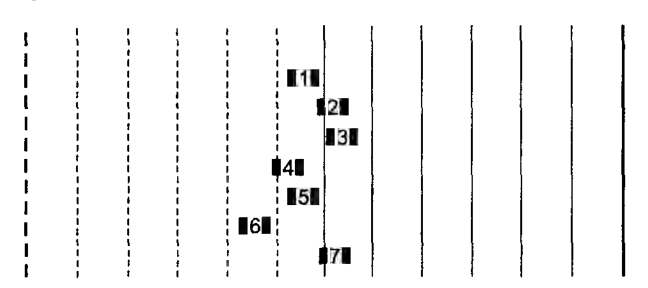


Рис. 10.19. Работа программы HORSE

Разработка лошадиных скачек

Как мы можем применить ООП для нашей программы? Первый вопрос заключается в том, существует ли в нашей программе группа похожих объектов, с которыми мы будем работать? Ответ положительный, это лошади. Кажется разумным представить каждую лошадь как объект. Мы создадим класс horse, в котором будут содержаться данные для каждой из лошадей, такие, как ее номер и дистанция, на которой эта лошадь была самой быстрой.

Однако у нас есть еще данные, которые имеют отношение к маршруту скачек. Они включают в себя длину дистанции, время, затраченное на ее прохождение в минутах и секундах (0:00 на старте), и общее количество лошадей. Поэтому имеет смысл создать объект, который будет одиночным членом класса track. На ум приходят и другие объекты, которые ассоциируются с лошадиными скачками, например жокеи или конюхи, но они не будут использоваться в нашей программе.

Существуют ли другие способы разработки программы? Например, использование наследования для того, чтобы сделать класс horse производным класса track? В нашем случае это не имеет смысла, так как лошади не являются чем-то, имеющим отношение к дистанции скачек; это совсем разные вещи. Другая возможность — переделать данные дистанции в статические данные класса horse. Однако обычно лучше использовать отдельный класс для каждой предметной области. Это выгоднее, так как позволяет упростить использования классов

для других целей, например при использовании класса track в автомобильных гонках.

Как же будут взаимодействовать объекты классов track и horse? (Или, в терминах UML, в чем состоит их связь?) Массив указателей на объекты класса horse может быть членом класса track, при этом track сможет иметь доступ к horse через эти указатели. При создании track будут созданы и horse. Track передаст каждому объекту класса horse указатель на себя, и они смогут иметь к нему доступ.

Вот листинг программы HORSE:

```
// horse.cpp
// модель лошадиных скачек
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
const int CPF = 5;
const int maxHorses = 7;
class track;
///////////////////////////////////////////////////////////
class horse
{
  private:
    const track* ptrTrack;
    const int horse_number;
    float finish_time;
    float distance_run;
  public:
    horse ( const int n, const track* ptrT ) :
      horse_number ( n ),
      ptrTrack ( ptrT ),
      distamce_run ( 0.0 )
      { }
    ~horse ( )
      { }
    void display_horse ( const float elapsed_time );
};
///////////////////////////////////////////////////////////
class track
{
  private:
    horse* hArray [ maxHorses ];
    int total_horses;
    int horse_count;
    const float track_length;
    float elapsed_time;
  public:
    track ( float lenT, int nH );
    ~track ( );
    void display_track ( );
    void run ( );
    float get_track_len ( ) const;
};
///////////////////////////////////////////////////////////
void horse::display_horse ( float elapsed_time )
{
```

```
  set_cursor_pos ( 1 + int ( distance_run * CPF ), 2 + horse_number * 2 );
  set_color ( static_cast<color>( cBLUE + horse_number ) );
  char horse_char = '0' + static_cast<char>( horse_number );
  putch ( ' ' ); putch ( '\xDB' ); putch ( horse_char ); putch ( '\xDB' );
  if ( distance_run < ptrTrack->get_track_len ( ) + 1.0 / CPF )
  {
    if ( rand ( ) % 3 )
      distance_run += 0.2F;
    finish_time = elapsed_time;
  }
  else
  {
    int mins = int ( finish_time ) / 60;
    int secs = int ( finish_time ) - mins * 60;
    cout << " Время = " << mins << ":" << secs;
  }
}
///////////////////////////////////////////////////////////
track::track ( float lenT, int nH ) :
  track_length ( lenT ),
  total_horses ( nH ),
  horse_count ( 0 ),
  elapsed_time ( 0.0 )
{
  init_graphics ( );
  total_horses = ( total_horses > maxHorses ) ? maxHorses : total_horses;
  for ( int j = 0; j < total_horses; j++ )
    hArray [ j ] = new horse ( horse_count++, this );

  time_t aTime;
  srand ( static_cast<unsigned>( time ( &aTime ) ) );
  display_track ( );
}
///////////////////////////////////////////////////////////
track::~track ( )
{
  for ( int j = 0; j < total_horses; j++ )
    delete hArray [ j ];
}
///////////////////////////////////////////////////////////
void track::display_track ( )
{
  clear_screen ( );

  for ( int f = 0; f <= track_length; f++ )
    for ( int r = 1; r <= total_horses * 2 + 1; r++ )
    {
      set_cursor_pos ( f * CPF + 5, r );
      if ( f == 0 || f == track_length )
        cout << '\xDE';
      else
        cout << '\xB3';
    }
}
///////////////////////////////////////////////////////////
void track::run ( )
```

```
{
  while ( !kbhit ( ) )
  {
    elapsed_time += 1.75;

    for ( int j = 0; j < total_horses; j++ )
      hArray [ j ]->display_horse ( elapsed_time );
    wait ( 500 );
  }
  getch ( );
  cout << endl;
}
///////////////////////////////////////////////////////////
float track::get_track_len ( ) const
  { return track_length; }
///////////////////////////////////////////////////////////
int main ( )
{
  float length;
  int total;

  cout << "\nВведите длину дистанции (от 1 до 12): ";
  cin >> length;
  cout << "\nВведите количество лошадей (от 1 до 7): ";
  cin >> total;
  track theTrack ( length, total );
  theTrack.run ( );

  return 0;
}
```

Моделирование хода времени

Программы-симуляторы обычно включают в себя действия, происходящие в определенный период времени. Для моделирования хода времени такие программы включают в себя фиксированные интервалы. В программе HORSE функция main() вызывает метод run() класса track. Он делает серию вызовов метода display_horse() внутри цикла while, по одному для каждой лошади. Этот метод предназначен для перемещения лошади в новую позицию на экране. Затем цикл while делает паузу на 500 миллисекунд с помощью функции wait(). Далее процесс повторяется до тех пор, пока скачки не будут окончены или пользователь не нажмет кнопку.

Уничтожение массива указателей на объекты

В конце программы деструктор класса track должен уничтожить объекты класса horse, которые были образованы конструктором класса track с использованием операции new. Заметим, что в этом случае мы не можем просто сделать так

```
delete [ ] hArray;
```

Эта запись удалит массив указателей, но не объекты, на которые они указывают. Поэтому мы должны вызвать каждый элемент массива и применить к нему операцию delete.

```
for ( int j = 0; j < total_horses; j++ )
  delete hArray [ j ];
```

Функция putch()

Мы хотим раскрасить всех лошадей в разные цвета, но не для всех компиляторов функция cout может генерировать цвет. В нашей версии Borland C++ Builder такая возможность есть. Но в любом случае в С есть функции, которые могут генерировать цвета. Для этой цели мы использовали функцию putch() при отображении лошадей на экран.

```
putch ( ' ' ); putch ( '\xDB' ); putch ( horse_char ); putch ( '\xDB' );
```

Эта функция требует включения файла CONIO.H (который входит в поставку компилятора). Но нам не надо подключать этот файл к файлу HORSE.CPP, так как он уже подключен к файлу MSOFTCON.H или BORLACON.H.

Диаграммы UML

Давайте рассмотрим диаграмму классов UML для программы HORSE. Она показана на рис. 10.20. На этой диаграмме представлена концепция UML, которая называется *многообразием.*

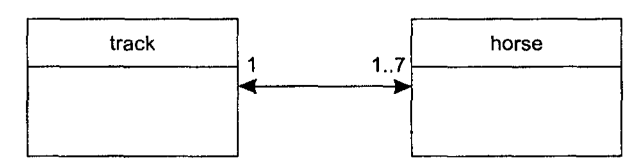


Рис. 10.20. UML-диаграмма классов программы HORSE

Иногда одному объекту класса A соответствует один объект класса B. В других ситуациях мы можем объединить несколько объектов класса. Это множество называется *многообразием.* Количество объектов, входящих в многообразие, обозначается на диаграмме с учетом табл. 10.2.

Таблица 10.2. Обозначения в многообразии UML

Символ	Значение
1	Один
*	Несколько (от 0 до бесконечности)
0..1	Один или ни одного
1..*	Хотя бы один
2..4	2, 3 или 4
7, 11	7 или 11

Если на диаграмме около класса A указано число 1, а возле класса B символ *, то это будет обозначать, что один объект класса A может взаимодействовать с произвольным количеством объектов класса B.

В программе HORSE с одним объектом класса track могут взаимодействовать до 7 объектов класса horse. Это обозначено цифрами 1 у класса track и 1..7 у класса horse. Мы предполагаем, что в скачках может принимать участие и одна лошадь, например во время тренировок.

Диаграмма состояний в UML

В этом разделе мы познакомимся с новым типом диаграмм UML: с *диаграммами состояний.*

Диаграммы классов UML мы рассматривали в предыдущих главах. На них были отражены *взаимоотношения* между классами. В диаграмме классов отражена организация кода программы. Это были *статические* диаграммы, в которых связи не изменяются при запуске программы.

Но иногда полезно рассмотреть объекты классов в *динамическом* режиме. С момента своего создания объект вовлекается в деятельность программы, выполняет различные действия и в конечном итоге удаляется. Ситуация постоянно изменяется, и это графически отражено на диаграмме состояний.

С концепцией *состояния* мы привыкли встречаться в нашей повседневной жизни. Радио, например, имеет два состояния: включенное и выключенное. Стиральная машина имеет такие состояния как стирка, полоскание, отжим и остановка. Для телевизора характерны состояния для каждого из его каналов (канал 7 включен и т. д.).

Между состояниями существуют *переходы.* Через 20 минут полоскания машина переходит в режим отжима. Получив сигнал от пульта управления, телевизор переключается из активного состояния канала 7 в активное состояние канала 2.

На рис. 10.21 показана диаграмма состояния для программы HORSE, рассмотренной ранее в этой главе. На ней отражены различные состояния объекта класса horse, которые он может принимать во время работы программы.

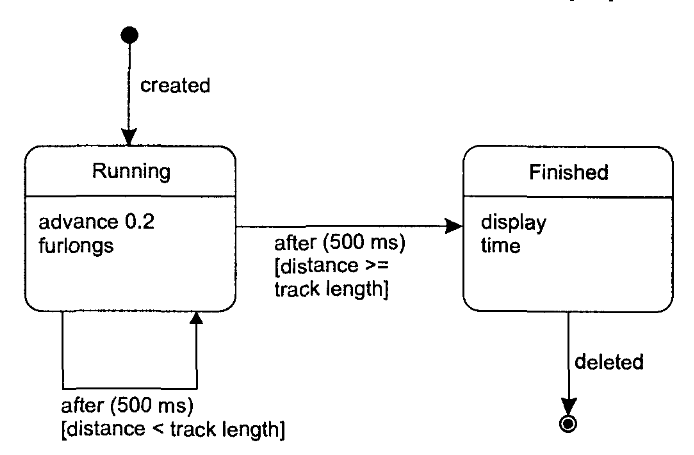


Рис. 10.21. Диаграмма состояний объекта класса horse

Состояния

В диаграммах состояний UML состояние представлено в виде прямоугольника со скругленными углами. Название состояния указано в верхней части прямоугольника, обычно оно начинается с заглавной буквы. Ниже указаны *действия*, которые выполняет объект, входя в это состояние.

На диаграмме присутствуют два специальных состояния: сплошным черным диском обозначено *начальное состояние*, а черным диском, помещенным в окружность, обозначено *конечное состояние*.

После создания объект класса horse может пребывать только в двух состояниях: до финишной линии — состояние Running и после ее достижения — состояние Finished.

В отличие от диаграмм классов, в коде программы нельзя точно указать фрагмент, соответствующий каждому конкретному состоянию. Чтобы разобраться в том, какие состояния должны входить в диаграмму, нужно представить ситуацию, в которой работает объект, и то, что мы хотим получить в результате. Затем каждому состоянию подбирают подходящее название.

Переходы

Переходы между состояниями представлены в диаграммах в виде стрелок, направленных от одного прямоугольника к другому. Если переход обусловлен каким-то событием, то он может быть обозначен его именем. В нашем случае так на рис. обозначены переходы created и deleted. Имя перехода не обозначают с заглавной буквы. Имена могут быть более приближены к реальному языку, чем к терминам С++.

Событие инициирует два других перехода по истечению периода времени 500 мс. Слово after использовано как имя для этих переходов с интервалом времени в качестве параметра.

Переходы могут быть также отмечены тем, что в UML называют *защитой*: это условие, которое должно быть выполнено для совершения перехода. Оно записывается в квадратных скобках. Оба перехода имеют защиту и имя события. Так как событие одинаковое, то условие защиты определяет, какой из переходов будет выполнен.

Заметим, что один из переходов является переходом *сам в себя*, он возвращает нас в то же состояние.

От состояния к состоянию

Каждый раз, попадая в состояние Running, объект класса horse выполняет действие, заключающееся в увеличении пройденного расстояния на 0.2 участка дистанции. Пока мы не достигли финиша, будет выполняться условие защиты [distance<track length], и мы будем возвращаться в состояние Running. Когда лошадь достигнет финиша, выполнится условие защиты [distance>=track length], и объект перейдет в состояние Finished, где будет выведено время скачки. Затем объект будет удален.

Мы показали достаточно, чтобы идея диаграмм состояния стала для вас понятной. Конечно, это не вся информация о них. Пример более сложной диаграммы состояния, описывающей объект класса elevator, мы рассмотрим в главе 13 «Многофайловые программы».

Отладка указателей

Указатели могут быть источником таинственных и катастрофических ошибок программы. Наиболее часто встречающейся ошибкой является указатель, которому не присвоили значение перед использованием. В этом случае указатель может указать на любой участок памяти. Это может быть участок, занимаемый кодом вашей программы или операционной системой. Если затем в память будет помещено некоторое значение с использованием этого указателя, то оно может записаться вне программы, и компьютер продемонстрирует вам весьма неожиданный результат.

Отдельной версией этой проблемы является случай, когда указатель указывает на адрес NULL. Например, это может случиться, если указатель определен как *глобальная переменная*, так как глобальные переменные автоматически инициализируются 0. Вот небольшая программа, демонстрирующая такую ситуацию:

```
int* intptr;    // глобальная переменная инициализируется нулем
void main ( )
{
  *intptr = 37; // нельзя поместить значение 37 по адресу
                // intptr, так как значение intptr равно
                // 0, а это является некорректным значением
                // для указателя
}
```

При определении указателю intptr будет присвоено значение 0, так как он является глобальной переменной. В строке программы происходит попытка поместить значение 37 по адресу NULL.

Однако встроенная в компилятор проверка ошибок при запуске остановится при попытке доступа к адресу NULL, выведет сообщение об ошибке (это может быть *нарушение доступа*, *присвоение значения нулевому адресу* или *ошибка из-за отсутствия страницы*) и остановит выполнение программы. Если вы увидите сообщения такого типа, то всегда существует вероятность того, что вы неправильно инициализировали указатель.

Резюме

Мы весьма поверхностно прошлись в этой главе по теме указателей. Она очень обширна, и в разделах этой главы мы рассмотрели лишь основные понятия, на которых построены примеры нашей книги и которых достаточно для дальнейшего изучения темы.

Мы узнали, что все объекты имеют адреса в памяти компьютера и что адреса — это *указатели константы*. Адрес переменной можно получить, используя операцию получения адреса &.

Указатели — это переменные, значением которых является адрес. Указатели определяются с использованием звездочки, которая означает *указатель на*. В определение указателя всегда включают тип данных, на который он указывает, так как компилятору необходимы сведения о том, на что будет указывать указатель и как правильно выполнять с ним арифметические действия. Доступ к тому, на что указывает указатель, мы получаем, применяя звездочку, которая в этом случае является *операцией разыменования*, то есть получения значения переменной, на которую указывает указатель.

Специальный тип void* означает указатель на любой тип данных. Он используется в ситуациях, когда один и тот же указатель должен хранить адреса переменных разных типов.

Доступ к элементу массива можно получить, используя операции массива с *квадратными скобками или указатели. Адрес массива является указателем-константой*, но его можно присвоить переменной-указателю, которая может быть увеличена или изменена.

Функция при передаче ей адреса переменной может работать с самой переменной (этого не происходит, когда аргумент передается по значению). В этом отношении передача аргумента по указателю дает то же преимущество, что и передача по ссылке, хотя для доступа к аргументам указателя нужно применять операцию *разыменования*. Однако указатели в некоторых случаях предоставляют более гибкий механизм.

Строковая константа может быть определена как массив или с использованием указателя; последний подход более гибок. Строки, представленные массивами типа char, обычно передаются в функции с использованием указателей.

Операция new предназначена для выделения требуемого количества памяти в системе. Она возвращает указатель на выделенный участок памяти. Используется для создания переменных и структур данных в течение исполнения программы. Операция delete освобождает память, выделенную с использованием операции new.

Если указатель указывает на объект, то доступ к членам объекта можно получить, используя операцию ->. Для доступа к членам структуры используется тот же синтаксис.

В классах и структурах могут содержаться элементы, которые являются указателями на сами эти классы или структуры. Это позволяет создать более сложные структуры, такие, как связные списки.

Могут существовать указатели на указатели. Такие переменные определяются с использованием двойной звездочки, например int** pptr.

Многообразие в диаграммах классов UML показывает количество объединенных объектов.

Диаграммы состояний UML показывают, как изменяются с течением времени ситуации, в которых находится объект. Состояния показываются на диаграммах в виде прямоугольников со скругленными углами, а переходы между состояниями — в виде прямых линий.

Вопросы

Ответы на эти вопросы вы сможете найти в приложении Ж.

1. Напишите выражение, которое выводит адрес переменной testvar.
2. Адреса смежных в памяти переменных типа float отличаются на ________.
3. Указатель — это:
   а) адрес переменной;
   б) обозначение переменной, которая будет доступна следующей;
   в) переменная для хранения адреса;
   г) тип данных для адресных переменных.
4. Напишите выражения для:
   а) адреса переменной var;
   б) значения переменной, на которую указывает var;
   в) переменной var, используемой как аргумент по ссылке;
   г) типа данных указателя на char.
5. Адрес — это _________, а указатель это — __________.
6. Напишите определение для переменной указателя на float.
7. Указатели полезны при ссылке на адреса памяти, которые не имеют _____.
8. Пусть указатель testptr указывает на переменную testvar. Напишите выражение, которое позволит получить значение переменной testvar, не используя при этом ее имя.
9. Звездочка, расположенная после типа данных, означает _________. Звездочка, расположенная перед именем переменной, означает ___________.
10. Выражение *test означает:
    а) указатель на переменную test;
    б) ссылку на значение переменной test;
    в) разыменование переменной test;
    г) ссылку на значение переменной, на которую указывает test;
11. Является ли правильным следующий код?

```
int intvar = 333:
int* intptr:
cout << *intptr:
```

12. Указатель на void может содержать указатель на _____________.
13. В чем различие между intarr[3] и (intarr+3)?
14. Напишите код, который, используя указатели, выводит каждое значение массива intarr, имеющего 77 элементов.
15. Пусть intarr массив целых. Почему выражение intarr++ не правильно?
16. Из трех способов передачи аргумента в функцию только передача по ___ _______ и передача по ____________ позволяют функции изменять аргумент в вызывающей программе.

17. Тип переменной, на которую указывает указатель, должен присутствовать в определении указателя для того,
    а) чтобы типы данных не перемешались при выполнении арифметических операций;
    б) чтобы указатель мог быть использован для доступа к членам структуры;
    в) чтобы не было затронуто ни одно из религиозных убеждений;
    г) чтобы компилятор мог правильно выполнять арифметические операции и получать доступ к элементам массива.
18. Используя указатели, напишите прототип функции func(), которая возвращает значение типа void и принимает в качестве аргумента массив типа char.
19. Используя указатели напишите небольшую программу для перевода 80 символов строки s1 в строку s2.
20. Первый элемент строки это:
    а) имя строки;
    б) первый символ строки;
    в) длина строки;
    г) имя массива, содержащего строку.
21. Используя указатели, напишите прототип функции revstr(), которая возвращает строку и в качестве аргумента тоже принимает строку.
22. Запишите определение массива numptrs указателей на строки One, Two и Three.
23. Операция new:
    а) возвращает указатель на переменную;
    б) создает переменную с именем new;
    в) получает память для новой переменной;
    г) позволяет узнать, сколько памяти свободно на данный момент.
24. Использование операции new может привести к меньшим ____________ памяти, чем использование массива.
25. Операция delete возвращает ____________ операционной системе.
26. Пусть нам дан указатель p, указывающий на объект типа upperclass. Напишите выражение, позволяющее вызвать метод exclu() этого класса для данного объекта.
27. Пусть дан объект, являющийся элементом массива objarr под номером 7. Напишите выражение, которое позволит вызвать метод exclu() этого объекта.
28. Связный список — это:
    а) структура, где каждый элемент представляет собой указатель на следующий элемент;
    б) массив указателей, указывающих на элементы списка;

в) структура, в которой каждый элемент состоит из данных или указателя на данные;

г) структура, в которой элементы хранятся в массиве.

29. Напишите определение массива arr из 8 указателей, которые указывают на переменные типа float.

30. Если мы хотим отсортировать множество больших объектов или структур, то будет более эффективным:

а) поместить их в массив и сортировать как его элементы;

б) создать массив указателей на них и отсортировать его;

в) поместить эти объекты в связный список и отсортировать его;

г) поместить ссылки на эти объекты в массив и отсортировать его.

31. Изобразите многообразие объединений, которые имеют до 10 объектов с одной стороны и больше двух — с другой стороны.

32. Состояния в диаграмме состояний соответствуют:

а) сообщениям между объектами;

б) условиям, по которым объекты находят себя;

в) объектам программы;

г) изменениям ситуации, в которой используются объекты.

33. Истинно ли следующее утверждение: переходы между состояниями существуют во время исполнения программы?

34. Защита в диаграмме состояний — это:

а) ограничивающее условие на переход;

б) имя определенного перехода;

в) имя определенного состояния;

г) ограничение на создание определенных состояний.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Напишите программу, которая принимает группу чисел от пользователя и помещает их в массив типа float. После того как числа будут помещены в массив, программа должна подсчитать их среднее арифметическое и вывести результат на дисплей. Используйте указатели везде, где только возможно.

*2. Используйте класс String из примера NEWSTR этой главы. Добавьте к нему метод upit(), который будет переводить символы строки в верхний регистр. Вы можете использовать библиотечную функцию toupper(), которая принимает отдельный символ в качестве аргумента и возвращает символ, переведенный в верхний регистр (если это необходимо). Эта функция исполь-

зует заголовочный файл Cctype. Добавьте в функцию main() необходимые строки для тестирования метода upit().

*3. Используйте массив указателей на строки, представляющие собой названия дней недели, из примера PTROSTR этой главы. Напишите функции для сортировки этих строк в алфавитном порядке, используя в качестве основы функции bsort() и order() из программы PTRSORT этой главы. Сортировать необходимо указатели на строки, а не сами строки.

*4. Добавьте деструктор в программу LINKLIST. Он должен удалять все элементы списка при удалении объекта класса linklist. Элементы должны удаляться по очереди, в соответствии с их расположением в списке. Протестируйте деструктор путем вывода сообщения об удалении каждого из элементов списка; удалено должно быть также количество элементов, какое было положено в список (деструктор вызывается автоматически для каждого существующего объекта).

5. Предположим, что в функции main() определены три локальных массива одинакового размера и типа (скажем, float). Первые два уже инициализированы значениями. Напишите функцию addarrays(), которая принимает в качестве аргументов адреса трех массивов, складывает соответствующие элементы двух массивов и помещает результат в третий массив. Четвертым аргументом этой функции может быть размерность массивов. На всем протяжении программы используйте указатели.

6. Создайте свою версию библиотечной функции strcmp(s1, s2), которая сравнивает две строки и возвращает −1, если s1 идет первой по алфавиту, 0, если в s1 и s2 одинаковые значения, и 1, если s2 идет первой по алфавиту. Назовите вашу функцию compstr(). Она должна принимать в качестве аргументов два указателя на строки char*, сравнивать эти строки посимвольно и возвращать число int. Напишите функцию main() для проверки работы вашей функции с разными строками. Используйте указатели во всех возможных ситуациях.

7. Модифицируйте класс person из программы PERSORT этой главы так, чтобы он включал в себя не только имя человека, но и сведения о его зарплате в виде поля salary типа float. Вам будет необходимо изменить методы setName() и printName() на setData() и printData(), включив в них возможность ввода и вывода значения salary, как это можно сделать с именем. Вам также понадобится метод getSalary(). Используя указатели, напишите функцию salsort(), которая сортирует указатели массива persPtr по значениям зарплаты. Попробуйте вместить всю сортировку в функцию salsort(), не вызывая других функций, как это сделано в программе PERSORT. При этом не забывайте, что операция -> имеет больший приоритет, чем операция *, и вам нужно будет написать

```
if ( (*(pp+j))->getSalary() > (*(pp+k))->getSalary() )
  { /* меняем указатели местами */ }
```

8. Исправьте функцию additem() из программы LINKLIST так, чтобы она добавляла новый элемент в конец списка, а не в начало. Это будет означать,

что первый вставленный элемент будет выведен первым и результат работы программы будет следующим:

```
25
36
49
64
```

Для того чтобы добавить элемент, вам необходимо будет пройти по цепи до конца списка, а затем изменить указатель последнего элемента так, чтобы он указывал на новый элемент.

9. Допустим, что нам нужно сохранить 100 целых чисел так, чтобы иметь к ним легкий доступ. Допустим, что при этом у нас есть проблема: память нашего компьютера так фрагментирована, что может хранить массив, наибольшее количество элементов в котором равно десяти (такие проблемы действительно появляются, хотя обычно это происходит с объектами, занимающими большое количество памяти). Вы можете решить эту проблему, определив 10 разных массивов по 10 элементов в каждом и массив из 10 указателей на эти массивы. Массивы будут иметь имена a0, a1, a2 и т. д. Адрес каждого массива будет сохранен в массиве указателей типа int*, который называется ap. Вы сможете получить доступ к отдельному целому используя выражение ap[j] [k], где j является номером элемента массива указателей, а k — номером элемента в массиве, на который этот указатель указывает. Это похоже на двумерный массив, но в действительности является группой одномерных массивов.

   Заполните группу массивов тестовыми данными (скажем, номерами 0, 10, 20 и т. д.), а затем выведите их, чтобы убедиться, что все работает правильно.

10. Описанный в упражнении 9 подход нерационален, так как каждый из 10 массивов объявляется отдельно, с использованием отдельного имени, и каждый адрес получают отдельно. Вы можете упростить программу, используя операцию new, которая позволит вам выделить память для массивов в цикле и одновременно связать с ними указатели:

```
for ( j = 0; j < NUMARRAYS; j++ )      // создаем NUMARRAYS массивов
  *( ap + j ) = new int [ MAXSIZE ]; // по MAXSIZE целых чисел в каждом
```

   Перепишите программу упражнения 9, используя этот подход. Доступ к отдельному элементу массивов вы сможете получить, используя то же выражение, что и в упражнении 9, или аналогичное выражение с указателями: *(*(ap+j)+k).

11. Создайте класс, который позволит вам использовать 10 отдельных массивов из упражнения 10 как один одномерный массив, допуская применение операций массива. То есть мы можем получить доступ к элементам массива, записав в функции main() выражение типа a[j], а методы класса могут получить доступ к полям класса, используя двухшаговый подход. Перегрузим операцию [ ] (см. главу 9 «Наследование»), чтобы получить нужный нам результат. Заполним массив данными и выведем их. Хотя

для интерфейса класса использованы операции индексации массива, вам следует использовать указатели внутри методов класса.

12. Указатели сложны, поэтому давайте посмотрим, сможем ли мы сделать работу с ними более понятной (или, возможно, более непонятной), используя их симуляцию в классе.

Для разъяснения действия наших доморощенных указателей мы смоделируем память компьютера с помощью массивов. Так как доступ к массивам всем понятен, то вы сможете увидеть, что реально происходит, когда мы используем для доступа к памяти указатели.

Мы будем использовать один массив типа char для хранения всех типов переменных. Именно так устроена память компьютера: массив байтов (тип char имеет тот же размер), каждый из которых имеет адрес (или, в терминах массива, индекс). Однако C++ не позволит нам хранить данные типа float или int в массиве типа char обычным путем (мы можем использовать объединения, но это другая история). Поэтому мы создадим симулятор памяти, используя отдельный массив для каждого типа данных, которые мы хотим сохранить. В этом упражнении мы ограничимся одним типом float, и нам понадобится массив для него. Назовем этот массив fmemory. Однако значения указателей (адреса) тоже хранятся в памяти, и нам понадобится еще один массив для их хранения. Так как в качестве модели адресов мы используем индексы массива, то нам потребуется массив типа int, назовем его pmemory, для хранения этих индексов.

Индекс массива fmemory (назовем его fmem_top) показывает на следующее по очереди доступное место, где можно сохранить значение типа float. У нас есть еще похожий индекс массива pmemory (назовем его pmem_top). Не волнуйтесь о том, что наша «память» может закончиться. Мы предполагаем, что эти массивы достаточно большие, чтобы хранить все, что мы захотим, и нам не надо заботиться об управлении памятью.

Создадим класс Float, который мы будем использовать для моделирования чисел типа float, которые будет храниться в fmemory вместо настоящей памяти. Класс Float содержит поле, значением которого является индекс массива fmemory, хранящего значения типа float. Назовем это поле addr. В классе также должны быть два метода. Первый — это конструктор, имеющий один аргумент типа float для инициализации значения. Конструктор помещает значение аргумента в элемент массива fmemory, на который указывает указатель fmem_top, а затем записывает значение fmem_top в массив addr. Это похоже на то, как компоновщик и компилятор хранят обычные переменные в памяти. Второй метод является перегружаемой операцией &. Он просто возвращает значение указателя (индекса типа int) в addr.

Создадим второй класс ptrFloat. Объект этого класса содержит адрес (индекс) в pmemory. Метод класса инициализирует этот «указатель» значением типа int. Второй метод перегружает операцию * (*операция разыменования*). Его действия более сложны. Он получает адрес из массива pmemory, в котором хранятся адреса. Затем полученный адрес используется как ин-

декс массива fmemory для получения значения типа float, которое располагалось по нужному нам адресу.

```
float& ptrFloat::operator* ( )
{
  return fmemory [ pmemory [ addr ] ];
}
```

Таким образом мы моделируем действия операции разыменования (*). Заметим, что вам нужно возвращаться из этой функции по ссылке, чтобы можно было использовать операцию * слева от знака равно.

Классы Float и ptrFloat похожи, но класс Float хранит данные типа float в массиве, представляющем собой память, а класс ptrFloat хранит поля типа int (являющиеся у нас указателями, но на самом деле индексами массива) в другом массиве, который тоже представляет собой память.

Это типичное использование этих классов в функции main():

```
Float var1 = 1.234;           // определяем и инициализируем
Float var2 = 5.678;           // две вещественные переменные

ptrFloat ptr1 = &var1;        // определяем и инициализируем
ptrFloat ptr2 = &var2;        // два указателя

cout << " *ptr1 = " << *ptr1; // получаем значения переменных
cout << " *ptr2 = " << *ptr2; // и выводим на экран

*ptr1 = 7.123;                // присваиваем новые значения
*ptr2 = 8.456;                // переменным, адресованным через указатели

cout << " *ptr1 = " << *ptr1; // снова получаем значения
cout << " *ptr2 = " << *ptr2; // и выводим на экран
```

Заметим, что за исключением других имен типов переменных, это выглядит так же, как действия с настоящими переменными. Далее результат работы программы:

```
*ptr1 = 1.234
*ptr2 = 5.678

*ptr1 = 7.123
*ptr2 = 8.456
```

Такой путь реализации указателей может показаться очень сложным, но здесь показана их внутренняя работа и работа операции адреса. Мы рассмотрели природу указателей в различных ракурсах.

Глава 11
Виртуальные функции

- ✦ Виртуальные функции
- ✦ Дружественные функции
- ✦ Статические функции
- ✦ Инициализация копирования и присваивания
- ✦ Указатель this
- ✦ Динамическая информация о типах

Теперь, когда мы уже знаем кое-что об указателях, перейдем к изучению более сложных вопросов C++. Эта глава посвящена таким довольно слабо связанным между собой темам, как виртуальные функции, дружественные функции, статические функции, перегружаемые операции и методы, а также указатель this. Все это — прогрессивные особенности языка, однако далеко не везде их стоит использовать. Небольшие программы могут вполне обойтись и без них. Тем не менее они часто применяются и очень важны для больших, серьезных программ. В частности, виртуальные функции необходимы при использовании полиморфизма — одного из краеугольных камней ООП.

Виртуальные функции

Виртуальный означает *видимый, но не существующий в реальности*. Когда используются виртуальные функции, программа, которая, казалось бы, вызывает функцию одного класса, может в этот момент вызывать функцию совсем другого класса. А для чего вообще нужны виртуальные функции? Представьте, что имеется набор объектов разных классов, но вам хочется, чтобы они все были в одном массиве и вызывались с помощью одного и того же выражения. Например, в графической программе MULTSHAP из главы 9 «Наследование» есть разные геометрические фигуры: треугольник, шар, квадрат и т. д. В каждом из этих классов есть функция draw(), которая прорисовывает на экране фигуры.

Теперь, допустим, вам захотелось создать картинку, сгруппировав некоторые из этих элементов. Как бы сделать это без лишних сложностей? Подход к реше-

нию этой задачи таков: создайте массив указателей на все неповторяющиеся элементы картинки:

```
shape* ptarr[100]; //массив из 100 указателей на фигуры
```

Если в этом массиве содержатся указатели на все необходимые геометрические фигуры, то вся картинка может быть нарисована в обычном цикле:

```
for(int j=0; j<N; j++)
    ptarr[j]->draw();
```

Это просто потрясающая возможность! Абсолютно разные функции выполняются с помощью одного и того же вызова! Если указатель в массиве ptarr указывает на шарик, вызовется функция, рисующая шарик, если он указывает на треугольник, то рисуется треугольник. Вот это и есть *полиморфизм*, то есть *различные формы*. Функции выглядят одинаково — это выражение draw(), но реально вызываются разные функции, в зависимости от значения ptarr[j]. Полиморфизм — одна из ключевых особенностей объектно-ориентированного программирования (ООП) наряду с классами и наследованием.

Чтобы использовать полиморфизм, необходимо выполнять некоторые условия. Во-первых, все классы (все эти треугольнички, шарики и т. д.) должны являться наследниками одного и того же базового класса. В MULTSHAP этот класс называется shape. Во-вторых, функция draw() должна быть объявлена виртуальной (virtual) в базовом классе.

Все это выглядит абстрактно, поэтому давайте напишем несколько небольших программ, которые выявят некоторые практические вопросы, чтобы потом можно было собрать их воедино.

Доступ к обычным методам через указатели

Наш первый пример покажет, что бывает, когда базовый и производные классы содержат функции с одним и тем же именем, и к ним обращаются с помощью указателей, но без использования виртуальных функций.

Листинг 11.1. Программа NOTVIRT[1]

```
// notvirt.cpp
// Доступ к обычным функциям через указатели
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Base              //Базовый класс
    {
    public:
  void  show()          //Обычная функция
    {  cout << "Base\n"; }
    };
//////////////////////////////////////////////////////////
class Dervl : public Base //Производный класс 1
    {
    public:
```

продолжение

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

Листинг 11.1 (*продолжение*)

```
   void  show()
      {  cout << "Derv1\n"; }
      };
/////////////////////////////////////////////
class Derv2 : public Base //Производный класс 2
      {
      public:
   void  show()
      {  cout << "Derv2\n"; }
      };
/////////////////////////////////////////////
int main()
      {
      Derv1 dv1;          //Объект производного класса 1
      Derv2 dv2;          //Объект производного класса 2
      Base* ptr;          //Указатель на базовый класс

      ptr = &dv1;         //Адрес dv1 занести в указатель
      ptr->show()         //Выполнить show()
      ptr = &dv2;         //Адрес dv2 занести в указатель
      ptr->show()         //Выполнить show()
      return 0;
      }
```

Итак, классы Derv1 и Derv2 являются наследниками класса Base. В каждом из этих трех классов имеется метод show(). В main() мы создаем объекты классов Derv1 и Derv2, а также указатель на класс Base. Затем адрес объекта порожденного класса мы заносим в указатель базового класса:

```
ptr = &dv1;  //Адрес объекта порожденного класса заносим в //указатель базового
```

Но постойте, а компилятор на нас не обидится за то, что мы присваиваем адрес объекта одного типа указателю на другой тип? Оказывается, наоборот — компилятор будет просто счастлив, потому что проверка типов отдыхает в этой ситуации. Мы скоро поймем, почему. Дело в том, что указатели на объекты порожденных классов совместимы по типу с указателями на объекты базового класса.

Теперь хорошо бы понять, какая же, собственно, функция выполняется в этой строчке:

```
ptr->show();
```

Это функция Base::show() или Derv1::show()? Опять же, в последних двух строчках программы NOTVIRT мы присвоили указателю адрес объекта, принадлежащего классу Derv2, и снова выполнили

```
ptr->show();
```

Так какая же из функций show() реально вызывается? Результат выполнения программы дает простой ответ:

```
Base
Base
```

Как видите, всегда выполняется метод базового класса. Компилятор не смотрит на *содержимое* указателя ptr, а выбирает тот метод, который удовлетворяет *типу* указателя, как показано на рис. 11.1.

Да, иногда именно это нам и нужно, но таким образом не решить поставленную в начале этой темы проблему доступа к объектам разных классов с помощью одного выражения.

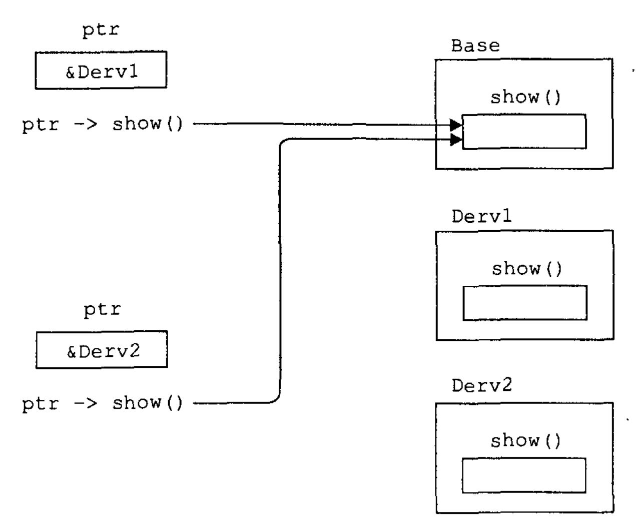


Рис. 11.1. Доступ через указатель без использования виртуальных функций

Доступ к виртуальным методам через указатели

Давайте сделаем одно маленькое изменение в нашей программе: поставим ключевое слово virtual перед объявлением функции show() в базовом классе. Вот листинг того, что получилось — программы VIRT:

Листинг 11.2. Программа VIRT

```
// virt.cpp
// Доступ к виртуальным функциям через указатели
#include <iostream>
using namespace std:
///////////////////////////////////////////////
class Base                  //Базовый класс
   {
   public:
  virtual void  show()    //Виртуальная функция
     {  cout << "Base\n": }
   }:
///////////////////////////////////////////////
class Derv1 : public Base //Производный класс 1
   {
   public:
  void  show()
     {  cout << "Derv1\n": }
   }:
///////////////////////////////////////////////
class Derv2 : public Base //Производный класс 2
```

продолжение

Листинг 11.2 (*продолжение*)

```
  {
  public:
 void  show()
    {  cout << "Derv2\n"; }
  };
//////////////////////////////////////////////
int main()
  {
  Derv1 dv1;         //Объект производного класса 1
  Derv2 dv2;         //Объект производного класса 2
  Base* ptr;         //Указатель на базовый класс

  ptr = &dv1;        //Адрес dv1 занести в указатель
  ptr->show()        //Выполнить show()
  ptr = &dv2;        //Адрес dv2 занести в указатель
  ptr->show()        //Выполнить show()
  return 0;
  }
```

На выходе имеем:

```
Derv1
Derv2
```

Теперь, как видите, выполняются методы производных классов, а не базового. Мы изменили содержимое ptr с адреса из класса Derv1 на адрес из класса Derv2, и изменилось реальное выполнение show(). Значит, один и тот же вызов

```
ptr->show()
```

ставит на выполнение разные функции в зависимости от содержимого ptr. Компилятор выбирает функцию, удовлетворяющую тому, что занесено в указатель, а не типу указателя, как было в программе NOTVIRT. Это показано на рис. 11.2.

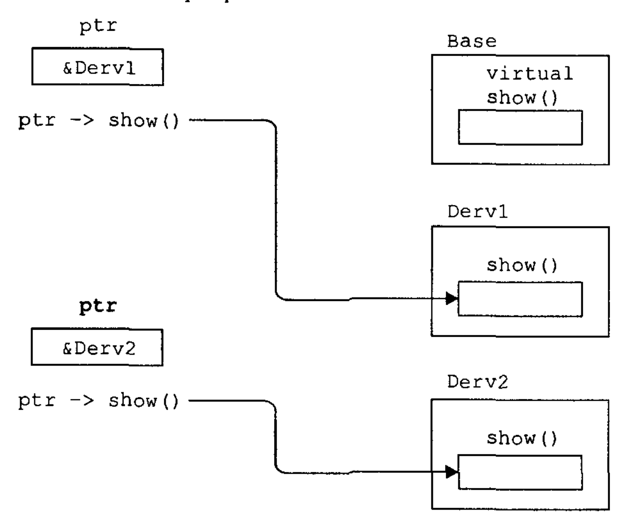


Рис. 11.2. Доступ через указатель к виртуальным функциям

Позднее связывание

Любознательный читатель может удивиться, как же компилятор узнает, какую именно функцию ему компилировать? В программе NOTVIRT у компилятора нет проблем с выражением

```
ptr->show();
```

Он всегда компилирует вызов функции show() из базового класса. Однако в программе VIRT компилятор не знает, к какому классу относится содержимое ptr. Ведь это может быть адрес объекта как класса Derv1, так и класса Derv2. Какую именно версию draw() вызывает компилятор — тоже загадка. На самом деле компилятор не очень понимает, что ему делать, поэтому откладывает принятие решения до фактического запуска программы. А когда программа уже поставлена на выполнение, когда известно, на что указывает ptr, тогда будет запущена соответствующая версия draw. Такой подход называется *поздним связыванием* или *динамическим связыванием*. (Выбор функций в обычном порядке, во время компиляции, называется *ранним связыванием* или *статическим связыванием*.) Позднее связывание требует больше ресурсов, но дает выигрыш в возможностях и гибкости.

Вскоре мы претворим эти идеи в жизнь, а сейчас вернемся к виртуальным функциям.

Абстрактные классы и чистые виртуальные функции

Давайте вернемся к классу shape из программы MULTSHAP (глава 9). Мы никогда не станем создавать объект из класса shape, разве что начертим несколько геометрических фигур — кругов, треугольников и т. п. Базовый класс, объекты которого никогда не будут реализованы, называется *абстрактным классом*. Такой класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы. Еще он может служить звеном для создания иерархической структуры классов.

Как нам объяснить людям, использующим созданную нами структуру классов, что объекты родительского класса не предназначены для реализации? Можно, конечно, заявить об этом в документации, но это никак не защитит наш базовый класс от использования не по назначению. Надо защитить его программно. Для этого достаточно ввести в класс хотя бы одну *чистую виртуальную функцию*. Чистая виртуальная функция — это функция, после объявления которой добавлено выражение =0. Продемонстрируем сказанное в примере VIRTPURE:

Листинг 11.3. Программа VIRTPURE

```
// virtpure.cpp
// Чистая виртуальная функция
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
```

продолжение ⇒

Листинг 11.3 (*продолжение*)

```
class Base                  //базовый класс
  {
  public:
    virtual void show() = 0; //чистая виртуальная
                             //функция
  };
////////////////////////////////////////////////
class Dervl : public Base   //порожденный класс 1
  {
  public:
    void show()
      { cout << ?Dervl\n?; }
  };
////////////////////////////////////////////////
class Derv2 : public Base   //порожденный класс 2
  {
  public:
    void show()
      { cout << ?Derv2\n?; }
  };
////////////////////////////////////////////////
int main()
  {
// Base bad;                //невозможно создать обьект
                            //из абстрактного класса
  Base* arr[2];             //массив указателей на
                            //базовый класс
  Dervl dvl;                //Обьект производного класса 1
  Derv2 dv2;                //Обьект производного класса 2

  arr[0] = &dvl;            //Занести адрес dvl в массив
  arr[1] = &dv2;            //Занести адрес dv2 в массив

  arr[0]->show();           //Выполнить функцию show()
  arr[1]->show();           //над обоими объектами
  return 0;
  }
```

Здесь виртуальная функция show() объявляется так:

```
virtual void show() = 0;  //чистая виртуальная функция
```

Знак равенства не имеет ничего общего с операцией присваивания. Нулевое значение ничему не присваивается. Конструкция =0 — это просто способ сообщить компилятору, что функция будет чистой виртуальной. Если теперь в main() попытаться создать объект класса Base, то компилятор будет недоволен тем, что объект абстрактного класса пытаются реализовать. Он выдаст даже имя чистой виртуальной функции, которая делает класс абстрактным. Помните, что хотя это только объявление, но определение функции show() базового класса не является обязательным. Впрочем, если вам надо его написать, это можно сделать.

Как только в базовом классе окажется чистая виртуальная функция, необходимо будет позаботиться о том, чтобы избежать ее употребления во всех производных классах, которые вы собираетесь реализовать. Если класс использует чистую виртуальную функцию, он сам становится абстрактным, никакие объекты

из него реализовать не удастся (производные от него классы, впрочем, уже не имеют этого ограничения). Более из эстетических соображений, нежели из каких-либо иных, можно все виртуальные функции базового класса сделать чистыми.

Между прочим, мы внесли еще одно, не связанное с предыдущими, изменение в VIRTPURE: теперь адреса методов хранятся в массиве указателей и доступны как элементы этого массива. Обработка этого, впрочем, ничем не отличается от использования единственного указателя. VIRTPURE выдает результат, не отличающийся от VIRT:

```
Derv1
Derv2
```

Виртуальные функции и класс person

Теперь, уже зная, что такое виртуальные функции, рассмотрим области их применения. Примером будет служить расширение программ PTROBJ и PERSORT из главы 10 «Указатели». Новая программа использует все тот же класс person, но добавлены два новых класса: student и professor. Каждый из этих производных классов содержит метод isOutstanding(). С помощью этой функции администрация школы может создать список выдающихся педагогов и учащихся, которых следует наградить за их успехи. Листинг программы VIRTPERS:

Листинг 11.4. Программа VIRTPERS

```
// vitrpers.cpp
// виртуальные функции и класс person
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class person                      //класс person
   {
   protected:
     char name[40];
   public:
     void getName()
       { cout << "  Введите имя: "; cin >> name; }
     void putName()
       { cout << "  Имя: " << name << endl; }
     virtual void getData() = 0;      //чистые
     virtual bool isOutstanding() =0; //виртуальные
                                      //функции
   };
////////////////////////////////////////////////
class student : public person   //класс student
   {
   private:
     float gpa;                   //средний балл
   public:
     void getData()               //запросить данные об ученике у
       {                          //пользователя
       person::getName();
       cout << "  Средний балл ученика:  "; cin >> gpa;
       }
```

продолжение ⇩

Листинг 11.4 (продолжение)

```
      bool isOutstanding()
        { return (gpa > 3.5) ? true : false; }
    };
////////////////////////////////////////////////
class professor : public person //класс professor
    {
    private:
      int numPubs;              //число публикаций
    public:
      void getData()            //запросить данные о педагоге у
        {                       //пользователя
        person::getName();
        cout << "  Число публикаций:  "; cin >> gpa;
        }
      bool isOutstanding()
        { return (gpa > 100) ? true : false; }
    };
////////////////////////////////////////////////
int main()
  {
  person* persPTR[100];  //массив указателей на person
   int n = 0;            //число людей, внесенных в список char choice;

  do {
    cout << " Учащийся (s) или педагог (p): ";
    cin >> choice;
    if(choice=='s' )              //Занести нового ученика
      persPtr[n] = new student;   // в массив
    else                          //Занести нового
      persPtr[n] = new professor; //педагога в массив
    persPtr[n++]->getData();      //Запрос данных о персоне
    cout << " Ввести еще персону (y/n)? "; //создать еще
                                           //персону
    cin >> choice;
    } while( choice=='y' );       //цикл, пока ответ 'y'

  for(int j=0; j<n; j++);
    {
    persPtr[j]->putName();        //Вывести все имена,
    if( persPtr[j]->isOutstanding() )      //сообщать о
      cout << "  Это выдающийся человек!\n"; //выдающихся
    }
  return 0;
  }                     //Конец main()
```

Классы

Класс person — абстрактный, так как содержит чистые виртуальные функции getData()и isOutstanding(). Никакие объекты класса person не могут быть созданы. Он существует только в качестве базового класса для student и professor. Эти порожденные классы добавляют новые экземпляры данных для базового класса. Student содержит переменную gpa типа float, представляющую собой средний балл учащегося. В классе Professor мы создали переменную numPubs типа int, которая представляет собой число публикаций педагога. Учащиеся со средним баллом свыше 3,5 и педагоги, опубликовавшие более 100 статей, считаются выда-

ющимися. (Мы, пожалуй, воздержимся от комментариев по поводу критериев оценки. Будем считать, что они выбраны условно.)

Функция isOutstanding()

IsOutstanding() объявлена чистой виртуальной функцией в классе person. В классе student эта функция возвращает true, если gpa больше 3,5, в противном случае — false. В классе professor она возвращает true, если numPubs больше 100.

Функция GetData() запрашивает у пользователя GPA, если она запускается для обслуживания класса student, или число публикаций для professor.

Программа main()

В функции main() мы вначале даем пользователю возможность ввести несколько имен учащихся и педагогов. К тому же программа спрашивает средний балл учащихся и число публикаций педагогов. После того, как пользователь закончит ввод данных, программа выводит на экран имена всех учащихся и педагогов, помечая выдающихся. Приведем пример работы программы:

```
Учащийся (s) или педагог (p): s
Введите имя: Сергеев Михаил
Средний балл ученика: 1.2
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): s
Введите имя: Пупкин Васипий
Средний балл ученика: 3.9
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): s
Введите имя: Борисов Впадимир
Средний балл ученика: 4.9
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): p
Введите имя: Михайлов Сергей
Число публикаций: 714
Ввести еще персону (y/n)? y
Учащийся (s) ипи педагог (p): p
Введите имя: Владимиров Борис
Число публикаций: 13
Ввести еще персону (y/n)? n

Имя: Сергеев Михаил
Имя: Пупкин Василий
Это выдающийся человек!
Имя: Борисов Владимир
Это выдающийся человек!
Имя: Михайлов Сергей
Это выдающийся человек!
Имя: Владимиров Борис
```

Виртуальные функции в графическом примере

Давайте рассмотрим еще один пример использования виртуальных функций. На сей раз это будет доработка программы MULTSHAP из главы 9 «Наследование». Как уже отмечалось в начале этой темы, вам может понадобиться нарисовать

некоторое число фигур с помощью одного и того же выражения. Это с успехом осуществляет программа VIRTSHAP. Помните, что компилировать эту программу следует, используя соответствующий файл для рисования графики, который описан в приложении Г «Упрощенный вариант консольной графики».

Листинг 11.5. Программа VIRTSHAP

```
// virtshap.cpp
// Виртуальные функции и геометрические фигуры
#include <iostream>
using namespace std;
#include "msoftcon.h"          //для графических функций
///////////////////////////////////////////////////////////
class shape                    //базовый класс
   {
   protected:
      int xCo, yCo;            //координаты центра
      color fillcolor;         //цвет
      fstyle fillstyle;        //заполнение
   public:                     //конструктор без аргументов
      shape() : xCo(0), yCo(0), fillcolor(cWHITE),
fillstyle(SOLID_FILL)
         { }                   //конструктор с четырьмя аргументами
      shape(int x, int y, color fc, fstyle fs) :
               xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
         { }
      virtual void draw() = 0 //чистая виртуальная функция
         {
         set_color(fillcolor);
         set_fill_style(fillstyle);
         }
   };
///////////////////////////////////////////////////////////
class ball : public shape
   {
   private:
      int radius;            //центр с координатами(xCo, yCo)
   public:
      ball() : shape()       //конструктор без аргументов
         { }
                             //конструктор с пятью аргументами
      ball(int x, int y, int r, color fc, fstyle fs)
               : shape(x, y, fc, fs), radius(r)
         { }
      void draw()            //нарисовать шарик
         {
         shape::draw();
         draw_circle(xCo, yCo, radius);
         }
   };
///////////////////////////////////////////////////////////
class rect : public shape
   {
   private:
      int width, height;   //(xCo, yCo) - верхний левый угол
   public:
      rect() : shape(), height(0), width(0) //конструктор //без аргументов
         { }                 //конструктор с шестью аргументами
    rect(int x, int y, int h, int w, color fc, fstyle fs) :
```

```
                     shape(x, y, fc, fs), height(h), width(w)
         {  }
      void draw()            //нарисовать прямоугольник
         {
         shape::draw();
         draw_rectangle(xCo, yCo, xCo+width, yCo+height);
         set_color(cWHITE);   //нарисовать диагональ
         draw_line(xCo, yCo, xCo+width, yCo+height);
         }
   };
///////////////////////////////////////////////////////////
class tria : public shape
   {
   private:
      int height;                //(xCo, yCo) - вершина пирамиды
   public:
      tria() : shape(), height(0) //конструктор без аргументов
         { }                     //конструктор с пятью аргументами
      tria(int x, int y, int h, color fc, fstyle fs) :
                              shape(x, y, fc, fs), height(h)
         {  }
      void draw()                //нарисовать треугольник
         {
         shape::draw();
         draw_pyramid(xCo, yCo, height);
         }
   };
///////////////////////////////////////////////////////////
int main()
   {
   int j;
   init_graphics();            //инициализация графики
   shape* pShapes[3];          //массив указателей на фигуры
                               //определить три фигуры
   pShapes[0] = new ball(40, 12, 5, cBLUE, X_FILL);
   pShapes[1] = new rect(12, 7, 10, 15, cRED, SOLID_FILL);
   pShapes[2] = new tria(60, 7,  11, cGREEN, MEDIUM_FILL);

   for(j=0; j<3; j++)          //нарисовать все фигуры
      pShapes[j]->draw();

   for(j=0; j<3; j++)          //удалить все фигуры
      delete pShapes[j];
   set_cursor_pos(1, 25);
   return 0;
   }
```

Спецификаторы классов в этой программе аналогичны спецификаторам из программы MULTSHAP, за исключением того, что draw() в классе shape стала чистой виртуальной функцией.

В main() мы задаем массив ptarr указателей на фигуры. Затем создаем три объекта, по одному из каждого класса, и помещаем их адреса в массив. Теперь с легкостью можно нарисовать все три фигуры:

```
ptarr[j]->draw();
```

Переменная j при этом меняется в цикле.

Как видите, это довольно мощный инструмент для соединения большого числа различных графических элементов в одно целое.

Виртуальные деструкторы

Знаете ли вы, что деструкторы базового класса обязательно должны быть виртуальными? Допустим, чтобы удалить объект порожденного класса, вы выполнили delete над указателем базового класса, указывающим на порожденный класс. Если деструктор базового класса не является виртуальным, тогда delete, будучи обычным методом, вызовет деструктор для базового класса вместо того, чтобы запустить деструктор для порожденного класса. Это приведет к тому, что будет удалена только та часть объекта, которая относится к базовому классу. Программа VIRTDEST демонстрирует это.

Листинг 11.6. Программа VIRTDEST

```
//vertdest.cpp
//Тест невиртуальных и виртуальных деструкторов
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////
class Base
   {
   public:
      ~Base()                  //невиртуальный деструктор
//    virtual ~Base()          //виртуальный деструктор
         { cout << "Base удален\n"; }
   };
/////////////////////////////////////////////////////////////
class Derv : public Base
  {
   public:
      ~Derv()
         { cout << "Derv удален\n"; }
   };
/////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

Программа выдаст такой результат:

```
Base удален
```

Это говорит о том, что деструктор для Derv не вызывается вообще! К такому результату привело то, что деструктор базового класса в приведенном листинге невиртуальный. Исправить это можно, закомментировав первую строчку определения деструктора и активизировав вторую. Теперь результатом работы программы является:

```
Derv удален
Base удален
```

Только теперь обе части объекта порожденного класса удалены корректно. Конечно, если ни один из деструкторов ничего особенно важного не делает (например, просто освобождает память, занятую с помощью new), тогда их виртуальность перестает быть такой уж необходимой. Но в общем случае, чтобы быть уверенным в том, что объекты порожденных классов удаляются так, как нужно, следует всегда делать деструкторы в базовых классах виртуальными.

Большинство библиотек классов имеют базовый класс, в котором есть виртуальный деструктор, что гарантирует нам наличие виртуальных деструкторов в порожденных классах.

Виртуальные базовые классы

Прежде чем расстаться с темой виртуальных элементов программирования, нам следует коснуться вопроса *виртуальных базовых классов*, так как они имеют отношение к множественному наследованию.

Рассмотрим ситуацию, представленную на рис. 11.3. Базовым классом является parent, есть два порожденных класса — child1, child2 и есть еще четвертый класс — Grandchild, порожденный одновременно классами Child1 и Child2.

В такой ситуации проблемы могут возникнуть, если метод класса Grandchild захочет получить доступ к данным или функциям класса Parent. Что в этом случае будет происходить, показано в программе NORMBASE.

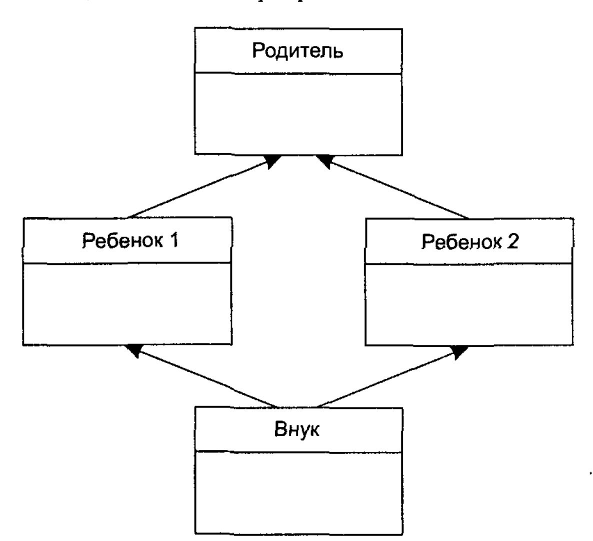


Рис. 11.3. Виртуальные базовые классы

Листинг 11.7. Программа NORMBASE

```
// normbase.cpp
// неоднозначная ссылка на базовый класс
class Parent
   {
   protected:
```

продолжение ⇨

Листинг 11.7 (*продолжение*)

```
      int basedata;
    };
class Child1 : public Parent
    { };
class Child2 : public Parent
    { };
class Grandchild : public Child1, public Child2
    {
    public:
      int getdata()
      { return basedata; }   // ОШИБКА: неоднозначность
    };
```

Ошибка компилятора возникла вследствие того, что метод Getdata() из класса Grandchild попытался получить доступ к basedata из класса parent. И что же в этом такого? Все дело в том, что каждый из порожденных классов (Child1 и Child2) наследует свою копию базового класса Parent. Эта копия называется *подобъектом*. Каждый из двух подобъектов содержит собственную копию данных базового класса, включая basedata. Затем, когда Grandchild ссылается на basedata, к какой из двух копий базового класса он получает доступ? Ситуация неоднозначная, о чем компилятор и сообщает.

Для устранения неоднозначности сделаем Child1 и Child2 наследниками виртуального базового класса, как показано в примере VIRTBASE.

Листинг 11.8. Программа VIRTBASE

```
// virtbase.cpp
// Виртуальные базовые классы
class Parent
    {
    protected:
      int basedata;
    };
class Child1 : virtual public Parent // наследует копию
                                     // класса Parent
    { };
class Child2 : virtual public Parent // наследует копию
                                     // класса Parent
    { };
class Grandchild : public Child1, public Child2
    {
    public:
      int getdata()
        { return basedata; }         // OK: только одна копия
                                     //класса Parent
    };
```

Использование ключевого слова virtual в этих двух классах приводит к тому, что они наследуют единый общий подобъект базового класса Parent. Так как теперь у нас есть только одна копия basedata, неоднозначность при обращении к базовому классу устраняется.

Необходимость использования виртуального базового класса показывает, что при множественном наследовании возникает ряд концептуальных проблем, поэтому такой подход должен использоваться осторожно.

Дружественные функции

Принцип инкапсуляции и ограничения доступа к данным запрещает функциям, не являющимся методами соответствующего класса, доступ к скрытым (private) или защищенным данным объекта. Политика этих принципов ООП такова, что, если функция не является членом объекта, она не может пользоваться определенным рядом данных. Тем не менее есть ситуации, когда такая жесткая дискриминация приводит к значительным неудобствам.

Дружественные функции как мосты между классами

Представьте, что вам необходимо, чтобы функция работала с объектами двух разных классов. Например, функция будет рассматривать объекты двух классов как аргументы и обрабатывать их скрытые данные. В такой ситуации ничто не спасет, кроме friend-функции. Приведем простой пример FRIEND, дающий представление о работе дружественных функций в качестве мостов между двумя классами.

Листинг 11.9. Программа FRIEND

```
// friend.cpp
// Дружественные функции
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class beta;              //нужно для объявления frifunc

class alpha
   {
   private:
     int data;
   public:
     alpha() : data(3) {  }          //конструктор без
                                     //аргументов
     friend int frifunc(alpha, beta); //дружественная
                                     //функция
   };
//////////////////////////////////////////////////////////
class beta
   {
   private:
     int data;
   public:
     beta() : data(7) {  }           //конструктор без
                                     //аргументов
friend int frifunc(alpha, beta);     //дружественная
                                     //функция
   };
//////////////////////////////////////////////////////////
int frifunc(alpha a, beta b)         //определение функции
```

продолжение

Листинг 11.9 (*продолжение*)

```
   {
   return( a.data + b.data );
   }
//--------------------------------------------------------
int main()
   {
   alpha aa;
   beta bb;

   cout << frifunc(aa, bb) << endl;   //вызов функции
   return 0;
   }
```

В этой программе мы видим два класса — alpha и beta. Конструкторы этих классов задают их единственные элементы данных в виде фиксированных значений (3 и 7 соответственно).

Нам необходимо, чтобы функция frifunc() имела доступ и к тем, и к другим скрытым данным, поэтому мы делаем ее дружественной функцией. Этой цели в объявлениях внутри каждого класса служит ключевое слово friend:

```
friend int frifunc(alpha, beta);
```

Это объявление может быть расположено где угодно внутри класса. Нет никакой разницы, запишем мы его в public- или в private-секцию.

Объект каждого класса передается как параметр функции frifunc(), и функция имеет доступ к скрытым данным обоих классов посредством этих аргументов. Функция выполняет не очень сложную работу по складыванию значений данных и выдаче их суммы. В main() осуществляется вызов этой функции и выводится на экран результат.

Напоминаем, что к классу нельзя обращаться до того, как он объявлен в программе. Объявление функции frifunc() (класс alpha) ссылается на класс beta, а значит, beta должен быть объявлен до alpha. Отсюда определение

```
class beta;
```

в начале программы.

Ломая стены

Надо отметить, что идея дружественных функций несколько сомнительна. Во время разработки C++ на эту тему велись споры и приводились аргументы против включения в язык такой возможности. С одной стороны, дружественные функции повышают гибкость языка, но, с другой стороны, они не соответствуют *принципу ограничения доступа к данным*, который гласит, что только функции-члены могут иметь доступ к скрытым данным класса.

Насколько серьезно противоречит единству концепции использование дружественных функций? Дружественная функция объявляется таковой в том классе, к данным которого она захочет впоследствии получить доступ. Таким образом, программист, не имеющий доступа к исходному коду класса, не может сделать функцию дружественной. В этом смысле целостность данных сохраняется.

Но все равно такие функции принципиально нечистоплотны и потенциально могут привести к «spaghetti-code» (код со слишком большим числом передач управления), когда множество дружественных функций фактически стирают границы между классами. По этой причине дружественные функции должны быть по крайней мере разбросаны по программе и встречаться как можно реже. Если вы обнаружили, что используете слишком много этих функций, стоит подумать об изменении программы.

Пример с английскими мерами длины

Иногда дружественные функции слишком удобны, чтобы от них отказываться. Наверное, самым общим примером является ситуация, когда они используются для повышения многосторонности перегружаемых операций. Следующая программа показывает ограниченность применения таких операторов без использования дружественных функций. Пример является вариацией на тему ENGLPLUS и ENGLCONV из главы 8 «Перегрузка операций». Программа называется NOFRI.

Листинг 11.10. Программа NOFRI

```
// nofri.cpp
// ограничение перегрузки оператора «+»
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////////
class Distance             //Класс английских расстояний
   {
   private:
     int feet;
     float inches;
   public:
     Distance() : feet(0), inches(0.0) //конструктор
                                       // (без аргументов)
     { }                               //конструктор (1 арг.)
     Distance(float fltfeet)           //переводит float в
                                       //Distance
     {                                 //feet - это целая часть
     feet = static_cast<int>(fltfeet);
       inches = 12*(fltfeet-feet);     //слева - дюймы
     }
   Distance(int ft, float in)        //конструктор (2 арг.)
     { feet = ft; inches = in; }
   void showdist()                   //вывод на экран расстояния
     { cout << feet << "\'-" << inches << '\"'; }
   Distance operator + (Distance);
   };
//----------------------------------------------------------
                    //прибавить расстояние к d2
Distance Distance::operator + (Distance d2) //сумма
   {
   int f = feet + d2.feet;        //добавить футы
   float i = inches + d2.inches;  //добавить дюймы
   if(i >= 12.0)                  //если сумма превышает 12.0,
     { i -= 12.0; f++; }          //уменьшить на 12 дюймов,
                                  //прибавить 1 фут
```

продолжение

Листинг 11.10 (*продолжение*)
```
   return Distance(f,i);         //Новый Distance с суммой
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance d1 = 2.5;           //конструктор переводит
   Distance d2 = 1.25;          //feet типа float в Distance
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;              //distance + float: OK
   cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;              //float + Distance: ОШИБКА
// cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

В этой программе оператор «+» переопределен на сложение двух объектов типа Distance. К тому же есть конструктор с одним аргументом, который переводит значение типа float, представляющее футы и десятые части футов, в значение типа Distance (то есть переводит числа из формата 10.25' в 10'-3").

Когда есть такой конструктор, можно написать такое выражение в main():

```
d3 = d1 + 10.0;
```

Перегружаемому + нужны объекты типа Distance и справа, и слева, но, найдя справа аргумент типа float, в соответствии с конструктором он переводит этот float в distance и осуществляет сложение.

Но вот что происходит при едва уловимом изменении в выражении:

```
d3 = 10.0 + d1;
```

Разве это выражение обрабатывается как следует? К сожалению, нет, так как объект, чьим методом является перегружаемый +, должен быть переменной, находящейся *слева* от оператора. Когда мы туда помещаем оператор другого типа или константу, компилятор использует + для сложения с этим типом (float в данном случае) вместо того, чтобы добавлять объекты типа Distance. Увы, этот оператор не умеет конвертировать float в Distance, поэтому не может выполнить сложение. Вот результат работы NOFRI:

```
d1 = 2'-6''
d2 = 1'-3''
d3 = 12'-6''
```

Второе сложение не будет скомпилировано, поэтому эти выражения закомментированы. Выйти из этой ситуации можно, создав новый объект типа Distance:

```
d3 = Distance(10, 0) + d1;
```

Но такое выражение ни очевидно, ни элегантно. Как написать нормальное, красивое выражение, которое могло бы содержать нужные типы как справа, так и слева от оператора? Как вы уже, наверное, догадались, нам поможет дружественная функция. Программа FRENGL демонстрирует, как именно.

Листинг 11.11. Программа FRENGL

```
// frengl.cpp
// Дружественная перегружаемая операция +
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class Distance                  //Класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
    Distance()                  //конструктор без аргументов
      { feet = 0; inches = 0.0; }
    Distance( float fltfeet ) //конструктор (1 арг.)
      {    //Переводит float в Distance
      feet = int(fltfeet);    //feet - целая часть
      inches = 12*(fltfeet-feet); //слева - дюймы
      }
    Distance(int ft, float in)    //конструктор (2 арг.)
      { feet = ft; inches = in; }
    void showdist()               //Вывести длину
      { cout << feet << "\'-" << inches << '\"';}
    friend Distance operator + (Distance, Distance); //дружественный
  };
//---------------------------------------------------------

Distance operator + (Distance d1, Distance d2) // d1 + d2
  {
  int f = d1.feet + d2.feet;       //+ футы
  float i = d1.inches + d2.inches; //+ дюймы
  if(i >= 12.0)                    //если больше 12 дюймов,
    { i -= 12.0; f++;  }           //уменьшить на 12 дюймов,
                                   //прибавить 1 фут
  return Distance(f,i);            //Новая длина с суммой
  }
//---------------------------------------------------------
int main()
  {
  Distance d1 = 2.5;          //конструктор переводит
  Distance d2 = 1.25;         //float-feet в Distance
  Distance d3;
  cout << "\nd1 = "; d1.showdist();
  cout << "\nd2 = "; d2.showdist();

  d3 = d1 + 10.0;             //distance + float: OK
  cout << "\nd3 = "; d3.showdist();
  d3 = 10.0 + d1;             //float + Distance: OK
  cout << "\nd3 = "; d3.showdist();
  cout << endl;
  return 0;
  }
```

Итак, перегружаемая операция + сделана дружественной:

```
friend Distance operator + (Distance, Distance);
```

Обратите внимание, что для перегружаемого + первый аргумент — метод, тогда как второй — дружественная функция. Будучи обычным методом, + оперировал одним из объектов как объектом, членом которого он был, а вторым — как аргументом. Став дружественным, + стал рассматривать оба объекта в качестве аргументов.

Единственное изменение, которое мы сделали, заключается в том, что переменные feet и inches, использовавшиеся в NOFRI для прямого доступа данных объекта, были заменены в FRENGL переменными d1.feet и d1.inches, так как этот объект теперь стал аргументом.

Помните, что для того, чтобы сделать функцию дружественной, только объявление функции внутри класса нужно начинать с ключевого слова friend. Определение класса пишется без изменений, как и вызов функции.

Дружественность и функциональная запись

Иногда дружественная функция позволяет использовать более понятный синтаксис вызова функции, чем обычный метод. Например, нам нужна функция, возводящая в квадрат экземпляр объекта типа Distance из приведенного выше примера и возвращающая значение в квадратных футах как тип float. Пример MISQ показывает, как это может быть сделано с помощью обычного метода.

Листинг 11.12. Программа MISQ

```
// misq.cpp
// метод square() для Distance
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance  //Класс английских длин
{
private:
int feet;
float inches;
public://конструктор (без аргументов)
Distance() : feet(0), inches(0.0)
{  }//constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{  }
void showdist() //показать длину
{ cout << feet << "\'-" << inches << '\"';}
float square();//обычный метод
};
//---------------------------------------------------------
float Distance::square()//возвращает квадрат
{//расстояния
float fltfeet = feet + inches/12; //перевод во float
float feetsqrd = fltfeet * fltfeet;  //возведение в //квадрат
return feetsqrd;//возвратить квадрат
}
///////////////////////////////////////////////////////////
int main()
{
```

```
Distance dist(3, 6.0)://конструктор с 2 арг. (3'-6")
float sqft;

sqft = dist.square()://вычислить квадрат расстояния
 //показать расст. и квадрат расст.
cout << "\nРасстояние = "; dist.showdist();
cout << "\nКвадрат расст. = " << sqft << " кв. футов\n";
return 0;
}
```

В main() программа создает значение Distance, возводит его в квадрат и выводит результат. Вот как это выглядит на экране:

```
Расстояние = 3'-6''
Квадрат расст. = 12.25 кв. футов
```

В main() мы использовали выражение

```
sqft = dist.square();
```

чтобы отыскать квадрат dist и присвоить его переменной sqft. Это выполняется корректно, но если мы все-таки хотим работать с объектами Distance, используя тот же синтаксис, что мы используем для работы с обычными числами, видимо, имеет смысл предпочесть функциональную запись:

```
sqft = square(dist);
```

Такого эффекта можно добиться, сделав square() дружественной функцией для класса Distance. Это показано в программе FRISQ.

Листинг 11.13. Программа FRISQ

```
// frisq.cpp
// Дружественная для Distance функция square()
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance                          //Класс английских длин
  {
  private:
    int feet;
    class Distance                      //Класс английских длин
  {
  private:
    int feet;
    float inches;
  public:
    Distance() : feet(0), inches(0.0) //конструктор (без
                                      //аргументов)
      { }
                                    //конструктор (2 аргумента)
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    void showdist()                 //вывести расстояние
      { cout << feet << "\'-" << inches << '\"'; }
    friend float square(Distance); //дружественная ф-ция
  };
//-----------------------------------------------------------
```

продолжение

Листинг 11.13 (*продолжение*)

```
float square(Distance d)              //вернуть квадрат
  {                                   //расстояния
  float fltfeet = d.feet + d.inches/12: //конвертировать
                                      //в тип float
  float feetsqrd = fltfeet * fltfeet;  //вычислить квадрат
  return feetsqrd:                     //вернуть квадратные футы
  }
///////////////////////////////////////////////////////////
int main()
  {
  Distance dist(3. 6.0):          //конструктор с двумя
                                  //аргументами (3'-6")
  float sqft:

  sqft = square(dist);            //вернуть квадрат dist
                         //Вывести расстояние и его квадрат
  cout << "\nРасстояние = ": dist.showdist():
  cout << "\nКвадрат расст. = " << sqft << " square feet\n":
  return 0;
  }
float inches;
  public:                   //конструктор (без аргументов)
    Distance() : feet(0). inches(0.0)
{  }                        //constructor (two args)
  Distance(int ft. float in) : feet(ft). inches(in)
      {  }
    void showdist()         //показать длину
      { cout << feet << "\'-" << inches << '\"':}
    float square():         //обычный метод
  }:
//--------------------------------------------------------
float Distance::square()            //возвращает квадрат
  {                                 //расстояния
  float fltfeet = feet + inches/12;  //перевод во float
  float feetsqrd = fltfeet * fltfeet: //возведение в
                                    //квадрат
  return feetsqrd:                  //возвратить квадрат
  }
///////////////////////////////////////////////////////////
int main()
  {
  Distance dist(3. 6.0);   //конструктор с двумя арг. (3'-6")
  float sqft:

  sqft = dist.square():    //вычислить квадрат расстояния
                           //показать расст. и квадрат расст.
  cout << "\nРасстояние = ": dist.showdist():
  cout << "\nКвадрат расст. = " << sqft << " кв. футов\n":
  return 0:
  }
```

Тогда как функция Square() в MISQ не требовала аргументов, будучи обычным методом, во FRISQ ей уже требуются аргументы. Вообще говоря, дружественная функция требует на один аргумент больше, чем метод. Функция square() во FRISQ похожа на аналогичную в MISQ, но она обращается к данным объекта Distance как d.inches или как d.feet, в отличие от обращения inches и feet.

Дружественные классы

Методы могут быть превращены в дружественные функции одновременно с определением всего класса как дружественного. Программа FRICLASS показывает, как это выглядит на практике.

Листинг 11.14. Программа FRICLASS

```
// friclass.cpp
// Дружественные классы
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class alpha
  {
  private:
    int data1;
  public:
    alpha() : data1(99) {  }  //конструктор
    friend class beta;        //beta - дружественный класс
  };
///////////////////////////////////////////////////////////
class beta
  {                          //все методы имеют доступ
  public:                    //к скрытым данным alpha
    void func1(alpha a)  { cout << "\ndata1=" << a.data1;}
    void func2(alpha a)  { cout << "\ndata1=" << a.data1;}
  };
///////////////////////////////////////////////////////////
int main()
  {
  alpha a;
  beta b;

  b.func1(a);
  b.func2(a);
  cout << endl;
  return 0;
  }
```

Для класса alpha весь класс beta провозглашен дружественным. Теперь все методы beta имеют доступ к скрытым данным класса alpha (в данной программе это лишь единственная переменная data1).

Обратите внимание: мы объявляем именно класс дружественным, используя выражение

```
friend class beta;
```

А можно было сделать и так: сначала объявить обычный класс beta:

```
class beta
```

затем объявить класс alpha, а внутри определения alpha объявить дружественность класса beta:

```
friend beta;
```

Статические функции

Если вы откроете главу 6 «Объекты и классы», то сможете найти там пример STATIC, в котором было дано представление о статических данных. Напоминаем, что статические данные не дублируются для каждого объекта. Скорее, один элемент данных используется всеми объектами класса. В том же примере был показан класс, который запоминал, сколько у него было объектов. Давайте попробуем расширить эту идею, показав, что функции, как и данные, могут быть статическими. Кроме демонстрации статических функций, наша программа создаст класс, который присваивает идентификационный номер (ID) каждому из своих объектов. Это позволяет спросить у объекта, кто он таков, что, кроме всего прочего, бывает полезно при отладке программ. Еще одна задача этого примера состоит в том, чтобы пролить свет на деструкторы и работу с ними.

Листинг 11.15. Программа STATFUNC

```
// statfunc.cpp
// Статические функции и ID объектов
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class gamma
  {
  private:
    static int total;        //всего объектов класса
                             //(только объявление)
    int id;                  //ID текущего объекта
  public:
    gamma()                  //конструктор без аргументов
      {
      total++;               //добавить объект
      id = total;            //id равен текущему значению total
      }
    ~gamma()                 //деструктор
      {
      total--;
      cout << "Удаление ID " << id  << endl;
      }
    static void showtotal() // статическая функция
      {
      cout << "Всего: " << total << endl;
      }
    void showid()            // Нестатическая функция
      {
      cout << "ID: " << id << endl;
      }
  };
//--------------------------------------------------------------
int gamma::total = 0;        // определение total
////////////////////////////////////////////////////////////////
int main()
  {
  gamma g1;
  gamma::showtotal();

  gamma g2, g3;
```

```
gamma::showtotal();

g1.showid();
g2.showid();
g3.showid();
cout << "----------конец программы----------\n";
return 0;
}
```

Доступ к статическим функциям

В приведенной программе у нас имеется статический элемент данных под именем total, принадлежащий классу gamma. В этой переменной хранится информация о том, сколько объектов создано в классе. Ее значение увеличивается конструктором и уменьшается деструктором.

Давайте предположим, что нам нужно иметь доступ к total, находясь вне класса. В листинге можно увидеть функцию showtotal(), показывающую значение этой переменной. Каким образом получить доступ к этой функции?

Когда некоторый элемент данных объявляется статическим, на весь класс выделяется единственная копия этого элемента, при этом не имеет значения, сколько объектов класса мы создали. На самом деле может вообще не быть никаких объектов, но нам интересно узнать даже об этом факте. Можно создать, конечно, некое чудо в перьях для вызова нужного метода, как в приведенных ниже строчках:

```
gamma dummyObj: //создать обьект только для того, чтобы //иметь возможность вызвать функцию
dummyObj.showtotal();   //Вызов функции
```

Но, согласитесь, это выглядит некрасиво. По правилам хорошего тона не следует обращаться к объекту, когда мы делаем что-либо, относящееся к классу в целом. Более целесообразно использовать просто имя класса и оператор явного задания функции:

```
gamma::showtotal();       //Так правильнее
```

Тем не менее, такой прием не сработает, если showtotal() — обычный метод. В этом случае, действительно, придется написать имя объекта, точку и название функции. Хотите иметь доступ к showtotal(), используя только лишь имя класса, — объявляйте ее статической. Это мы и проделали в STATFUNC, объявив

```
static void showtotal()
```

Объявленная таким образом функция будет иметь доступ просто через имя класса. Вот пример работы программы:

```
Всего: 1
Всего: 3
ID: 1
ID: 2
ID: 3
----------конец программы----------
Удаление ID 3
Удаление ID 2
Удаление ID 1
```

Здесь мы задаем один объект g1, после чего выводим значение total, оно равно 1. Потом создаем g2 и g3 и снова выводим значение total, на сей раз оно уже равно 3.

Нумерация объектов

В класс gamma мы поместили еще одну функцию, предназначенную для вывода идентификационных номеров его объектов. Эти ID положены равными переменной total в момент создания объекта, так что номер каждого объекта оказывается уникальным. Функция showID() выводит ID своего объекта. В нашей программе она вызывается трижды в выражениях

```
g1.showID();
g2.showID();
g3.showID();
```

Как показывают результаты работы программы, у каждого объекта имеется действительно свой уникальный номер. G1 пронумерован 1, g2 — 2, g3 — 3.

Деструкторы

После того как мы узнали, что объекты можно нумеровать, покажем одно интересное свойство деструкторов. STATFUNC выводит на экран сообщение «Конец программы», это написано в последней строчке кода. Но, как показывает пример работы программы, после вывода этой строчки еще кое-что происходит. А именно, три объекта, которые были созданы в процессе работы программы, должны быть удалены до ее завершения, чтобы восстановился доступ к памяти. Об этом заботится компилятор, выполняя деструктор.

Вставив в деструктор выражение, выводящее на экран сообщение, мы можем наблюдать за процессом его работы. Благодаря введенной нумерации, мы также можем наблюдать, в каком порядке удаляются объекты. Как видим, удаление происходит в порядке, обратном созданию. Можно предположить, что локальные объекты хранятся в стеке, и используется LIFO-подход.

Инициализация копирования и присваивания

Компилятор C++ всегда стоит на вашей стороне, выполняя рутинную работу, которой программиста незачем загружать. Если вы обязуетесь контролировать ситуацию, он может посчитаться с вашим мнением и предоставить некоторые вопросы решать самому, но по умолчанию компилятор все делает сам. Двумя чрезвычайно важными примерами этого процесса являются оператор присваивания и конструктор копирования.

Вы постоянно пользуетесь оператором присваивания, может быть, даже не задумываясь о том, что за ним стоит с точки зрения компилятора. Пусть a1 и a2 — некоторые объекты. Несмотря на то, что вы говорите компилятору: «присвоить a2 значение a1», выражение

```
a2 = a1;
```

заставит компилятор копировать данные из a1 элемент за элементом в a2. Таковы действия по умолчанию для оператора присваивания.

Вы, возможно, знакомы с инициализацией переменных. Инициализация одного объекта другим, как в выражении

```
alpha a2(a1):      //инициализировать a2 значением a1
```

вызывает подобные действия. Компилятор создает новый объект a2, затем элемент за элементом копирует данные из a1 в a2. Это то, что по умолчанию выполняет конструктор копирования.

Оба этих стандартных действия выполняются без проблем компилятором. Если поэлементное копирование из одного объекта в другой — это то, что вам нужно, то можно больше ничего не предпринимать в этом направлении. Тем не менее, если вы хотите научиться заставлять присваивание или инициализацию выполнять какие-либо более интеллектуальные действия, то придется разобраться в том, как обойти стандартную реакцию на эти операторы. Обсуждать технику перегрузки оператора присваивания и конструктора копирования мы будем отдельно друг от друга, затем соединим полученные знания в один пример, который дает возможность классу String более эффективно использовать память. Кроме этого, вы узнаете об объектной диаграмме UML.

Перегрузка оператора присваивания

Следующий коротенький пример демонстрирует технологию перегрузки оператора присваивания.

Листинг 11.16. Программа ASSIGN

```
// assign.cpp
// перегрузка операции присваивания (=)
#include <iostream>
using namespace std:
///////////////////////////////////////////////////////////
class alpha
  {
  private:
    int data:
  public:
    alpha()                    //конструктор без аргументов
      { }
    alpha(int d)               //конструктор с одним аргументом
      { data = d: }
    void display()             //вывести данные
      { cout << data: }
    alpha operator = (alpha& a) //перегружаемый =
      {
      data = a.data:           //не выполняется автоматически
      cout << "\nЗапущен оператор присваивания":
      return alpha(data):      //возвращает копию alpha
      }
  }:
///////////////////////////////////////////////////////////
```

продолжение

Листинг 11.16 (*продолжение*)
```
int main()
  {
  alpha a1(37);
  alpha a2;

  a2 = a1;                        //запуск перегружаемого =
  cout << "\na2="; a2.display(); //вывести a2

  alpha a3 = a2;                  //НЕ запускается =
  cout << "\na3="; a3.display(); //вывести a3
  cout << endl;
  return 0;
  }
```

Класс alpha очень прост, в нем содержится только один элемент данных. Конструкторы инициализируют данные, а методы выводят их значения на экран. Вот и все, что тут делается. А новизна примера ASSIGN заключается в функции operator=(), перегрузившей оператор = operator.

В main() мы определяем переменную a1 и присваиваем ей значение 37, определяем переменную a2, но значения ей не присваиваем. Затем, как видите, присваиваем a2 значение a1:

```
a2 = a1;          //Выражение присваивания
```

Тем самым мы запускаем нашу перегружаемую функцию operator=(). Программа ASSIGN заканчивается с таким результатом:

```
Запущен оператор присваивания
a2=37
a3=37
```

Инициализация — это не присваивание

В последних двух строчках кода ASSIGN мы инициализируем объект a3 объектом a2 и выводим его значение на экран. И пусть вас не смущает синтаксис в приведенном ниже выражении. Выражению

```
alpha a3 = a2;   //инициализация копирования, а не
                 //присваивание!
```

соответствует не присваивание, а инициализация. А эффект будет при этом тот же, что при использовании выражения

```
alpha a3(a2);    //альтернативный вариант инициализации
                 //копирования
```

Именно поэтому оператор присваивания выполняется только один раз, что и отражено в результатах работы программы в единственной строчке вызова:

```
Запущен оператор присваивания
```

Ответственность

Когда перегружается оператор присваивания, подразумевается, что за все, что по умолчанию делал этот оператор, отвечает программист. Часто это связано с копированием элементов данных из одного объекта в другой. Класс alpha из про-

граммы ASSIGN включает в себя только один элемент — data, поэтому функция operator=() копирует его значение с помощью следующего выражения:

```
data = a.data;
```

В обязанности этой функции также входит вывод на экран строки «Запущена операция присваивания», по которой мы можем определить, что функция выполняется.

Передача по ссылке

Аргумент для функции operator=() передается по ссылке. Нельзя сказать, что это совершенно необходимо, но обычно именно так и делается. Зачем? Как вы знаете, аргумент, передающийся по значению, создает в памяти собственную копию для функции, которой он передается. Аргумент, передающийся функции operator=(), не является исключением. А если объекты большие? Их копии могут занять в памяти очень много места, причем от них нет никакой пользы. Когда же значения аргументов передаются по ссылке, копии не создаются, что помогает экономить память.

Есть и еще одна причина. В определенных ситуациях необходимо отслеживать количество созданных объектов (как в примере STATFUNC, где мы нумеровали созданные объекты). Если же компилятор генерирует еще какие-то объекты всякий раз, когда используется оператор присваивания, то есть шанс сильно завысить оценку их количества. С помощью передачи по ссылке удается ликвидировать прецеденты ложного создания объектов.

Возврат значений

Как вы уже успели заметить, функция может возвращать результат в вызывающую программу по значению или по ссылке. При возвращении по значению происходит фактически передача результата, а значит, создается копия объекта, которая и возвращается в программу. В вызывающей программе этот вновь созданный объект может присваиваться какому-либо другому объекту или использоваться как-то еще, это уже не так важно. Если же возвращение происходит по ссылке, никакой новый объект не создается. Ссылка на исходный объект — вот и все, что возвращается в качестве результата.

Функция operator=() возвращает результат путем создания временного объекта alpha и его инициализации с помощью одноаргументного конструктора в выражении

```
return alpha(data)
```

Возвращаемое значение — это копия исходного, а не тот же самый объект, чьим методом является перегружаемая операция =. Такой стиль возвращения значений позволяет выстраивать эти функции в цепочку:

```
a3 = a2 = a1;
```

Да, впечатляет, но все равно возвращение результата по значению имеет все те же недостатки, что и передача по значению. Прежде всего, это создание в памяти копии возвращаемого объекта, что может привести к некоторой путанице.

А можем мы вернуть результат по ссылке, используя описатель, показанный ниже, для перегружаемого присваивания?

```
alpha& operator = (alpha& a)   //Неудачная мысль
```

К сожалению, нет, мы не можем осуществить возврат результата по ссылке при работе с локальными переменными данной функции.

Напомним, что локальные переменные, то есть созданные внутри функции и не объявленные статическими, уничтожаются при выходе из функции. Но, как мы знаем, возвращение по ссылке передает в вызывающую программу только адрес данных, который для локальных переменных указывает на данные, находящиеся внутри функции. При возврате из функции указатель хранит уже какое-то не имеющее смысла значение. Компилятор может сигнализировать о такой ситуации предупреждением. В разделе «Указатель this» этой главы мы покажем, как можно решить эту проблему.

Ограничение на наследование

Оператор присваивания уникален среди остальных операторов тем, что он не может наследоваться. Перегрузив присваивание в базовом классе, вы не сможете использовать ту же функцию в порожденных классах.

Конструктор копирования

Мы уже обсуждали, что можно определять объекты и одновременно их инициализировать другими с помощью двух видов выражений:

```
alpha a3(a2);        //инициализация копирования
alpha a3 = a2;       //инициализация копирования,
                     // альтернативная запись
```

Оба стиля определения включают в себя конструктор копирования, то есть конструктор, создающий новый объект и копирующий в него свои аргументы. По умолчанию этот конструктор производит поэлементное копирование. Это очень похоже на то, что делает оператор присваивания, с той лишь разницей, что конструктор создает новый объект.

Как и оператор присваивания, конструктор копирования может быть перегружен. В программе XOFXREF мы покажем, как это делается.

Листинг 11.17. Программа XOFXREF

```
// xofxref.cpp
// конструктор копирования: X(X&)
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////////
class alpha
  {
  private:
    int data;
  public:
    alpha()             //конструктор без аргументов
```

```
    { }
  alpha(int d)        //конструктор с одним аргументом
    { data = d; }
  alpha(alpha& a)     //конструктор копирования
    {
    data = a.data;
    cout << "\nЗапущен конструктор копирования";
    }
  void display()      //display
    { cout << data; }
  void operator = (alpha& a) //overloaded = operator
    {
    data = a.data;
    cout << "\nЗапущен оператор присваивания";
    }
  };
///////////////////////////////////////////////////////////
int main()
  {
  alpha a1(37);
  alpha a2;

  a2 = a1;              //запуск перегружаемого =
  cout << "\na2="; a2.display();  //вывести a2

  alpha a3(a1);         //запуск конструктора копирования
// alpha a3 = a1;       //эквивалентное определение a3
  cout << "\na3="; a3.display();  //вывести a3
  cout << endl;
  return 0;
  }
```

В этой программе перегружается и оператор присваивания, и конструктор копирования. Перегружаемое присваивание подобно аналогичному из программы ASSIGN. У конструктора копирования имеется один аргумент — объект типа **alpha**, переданный по ссылке. Вот его описатель:

```
alpha(alpha&)
```

Он имеет вид X(X&). Ниже приведен результат работы программы:

```
Запущен оператор присваивания
a2=37
```

Запущен конструктор копирования

```
a3=37
```

Выражение

```
a2 = a1;
```

запускает оператор присваивания, тогда как

```
alpha a3(a1);
```

запускает конструктор копирования. Для последней цели может быть использовано другое эквивалентное выражение:

```
alpha a3 = a1;
```

Как уже было показано, конструктор копирования может быть запущен во время определения объекта. Он также запускается, когда аргументы передаются функциям по значению и когда возвращается результат работы функций. Давайте кратко рассмотрим эти ситуации.

Аргументы функции

Итак, конструктор копирования запускается при передаче по значению аргумента функции. Он создает копию объекта, с которой функция и работает. Таким образом, если бы функция

```
void func(alpha);
```

была объявлена в XOFXREF и если бы она вызывалась выражением

```
func(a1);
```

то конструктор копирования создал бы копию объекта a1 для его использования func(). Само собой разумеется, что конструктор копирования не запускается при передаче аргумента по ссылке или при передаче указателя на объект. В этих случаях никаких копий не создается, и функция работает с исходной переменной.

Возврат результатов работы функции

Конструктор копирования также создает временный объект, когда значение возвращается из функции в основную программу. Допустим, что в XOFXREF была какая-либо подобная функция:

```
alpha func();
```

и она вызывалась выражением

```
a2 = func();
```

Конструктор копирования в этом случае запустился бы для того, чтобы создать копию возвращаемого функцией func() результата, а затем, после запуска оператора присваивания, этот результат был бы сохранен в a2.

Почему не просто X(X)?

А надо ли нам использовать ссылку в аргументе конструктора копирования? Может быть, можно просто передать значение? Но нет, компилятор сообщает о «выходе за пределы памяти» при попытке откомпилировать выражение

```
alpha(alpha a)
```

Почему? Да потому, что при передаче аргумента по значению создается его копия. Кто ее создает? Конструктор копирования. Но мы ведь и запускаем конструктор копирования, то есть он пытается запустить самого себя, поймать собственный хвост, до тех пор, пока не выйдет за пределы памяти. Так что в конструктор копирования аргумент должен передаваться по ссылке, что не приведет к созданию копий объекта.

Следите за деструкторами

В разделах «Передача по ссылке» и «Возврат значений» мы обсуждали передачу функциям аргументов по значению и возврат результатов в вызывающую программу по значению. В этих ситуациях также запускается деструктор, когда при выходе из функции уничтожаются временные объекты. Это может вызвать недоумение, если вы не ожидаете такого поворота событий. Мораль сей басни такова: работая с объектами, которым требуется нечто большее, нежели простое поэлементное копирование, передавайте и возвращайте данные по ссылкам везде, где это возможно.

Определение и конструктора копирования, и оператора присваивания

Перегружая оператор присваивания, вы почти наверняка намереваетесь заодно переопределить и конструктор копирования (и наоборот). Дело в том, что вам не всегда требуется использовать эти перегружаемые операции, но, с другой стороны, есть места в программах, где они необходимы. Даже если вы не предполагаете использовать в программе одно либо другое, можно заметить, что компилятор сам использует различные варианты поведения в сложных ситуациях, таких, как, например, передача аргумента по значению или возврат результата по значению.

Фактически, если конструктор класса включает в себя операции по работе с системными ресурсами — памятью, файлами на диске, — вам следует почти всегда перегружать и конструктор копирования, и оператор присваивания, не забывая убедиться в том, что они работают корректно.

Запрещение копирования

Мы обсудили, как настроить процедуру копирования объектов с помощью конструктора копирования и оператора присваивания. Но иногда бывает нужно запретить копирование. Сделать это можно, используя те же инструменты. Например, может быть важно то, что каждый член класса создается с уникальным значением определенного элемента данных, который поставляется в качестве аргумента конструктору. Но вы ведь понимаете, что если создастся копия объекта, ей будет присвоено то же значение, и объект перестанет быть уникальным. Чтобы избежать копирования, просто-напросто перегружайте присваивание и конструктор копирования, делайте их скрытыми членами класса:

```
class alpha
{
  private:
    alpha& operator = (alpha&); //Скрытое присваивание
    alpha(alpha&);              //Скрытое копирование
  };
```

Как только вы попытаетесь скопировать информацию

```
alpha a1, a2;
a1 = a2;                        //присваивание
alpha a3(a1);                   //копирование
```

компилятор сообщит, что функция недоступна. Вам нет никакой необходимости определять функции, которые никогда не будут вызваны.

Объектные диаграммы UML

В предыдущих главах мы видели примеры диаграмм классов. Наверное, не будет сюрпризом то, что UML поддерживает и объектные диаграммы. Эти диаграммы отображают некоторые определенные объекты (например, объект Mike_Gonzalez для класса Professor). Поскольку отношения между объектами могут меняться во время работы программы, объекты могут даже добавляться и исчезать, объектная диаграмма представляет собой некую мгновенную картину состояния объектов. Это называют *статической* диаграммой UML.

Объектную диаграмму можно использовать для моделирования какого-либо конкретного действия, выполняемого программой. Вы как бы приостанавливаете программу и смотрите на текущее состояние объектов в том аспекте, который вас интересует, и на соотношения объектов в этот момент времени.

В объектной диаграмме объекты представляются в виде прямоугольников как классы в диаграмме классов. Имя, атрибуты и операции отображаются аналогично. Отличить объекты от классов можно по тому, что их названия подчеркнуты. В строке названия можно использовать имя объекта и имя класса, разделенные двоеточием:

```
anObj:aClass
```

Если имя объекта вам неизвестно (например, потому что доступ к нему осуществляется только через указатель), можно использовать просто имя класса, начинающееся с двоеточия:

```
:aClass
```

Линии между объектами называются *связями* и отображают, разумеется, связь между объектами. Возможности навигации между объектами также могут быть показаны на диаграмме. Значение атрибута показывается с помощью знака равенства:

```
count = 0
```

Еще одним достоинством UML считаются *примечания*. Они показываются в прямоугольниках с загнутым уголком и содержат комментарии или пояснения. Пунктирная линия соединяет примечание с соответствующим элементом диаграммы. В отличие от связей и соединений, примечание может ссылаться на элемент внутри прямоугольника, содержащего класс или объект. Примечания могут использоваться в любых типах диаграмм UML.

В этой главе вы увидите довольно много объектных диаграмм.

Эффективное использование памяти классом String

Программы ASSIGN и XOFXREF на самом деле вовсе не нуждаются в перегрузке присваиваний и конструкторов копирования. Ведь они используют крайне простые

классы с единственным элементом данных, поэтому присваивание и копирование по умолчанию работали бы вполне нормально. Но сейчас мы рассмотрим пример, в котором крайне важно использовать перегрузку этих операторов.

Недостатки класса String

Мы уже насмотрелись на разные версии нашего кустарного класса String в предыдущих главах. И эти версии не являются на самом деле оптимальными. Было бы здорово, наверное, перегрузить оператор присваивания, тогда мы смогли бы присваивать значение одного объекта класса String другому с помощью выражения

```
s2 = s1;
```

Но, если мы и вправду перегрузим оператор присваивания, встанет вопрос, как работать с реальными строками (то есть массивами символьного типа char), которые являются принципиальными элементами данных для класса String.

Можно сделать так, чтобы каждый объект имел специально отведенное «место» для хранения строки. Тогда, присваивая один объект класса String другому, мы просто копируем строку из исходного объекта в объект назначения. Но если вы все же хотите экономить память, то необходимо обеспокоиться тем, чтобы одна и та же строка не дублировалась попусту в двух и более местах памяти. Поверьте, это не слишком эффективный подход, особенно учитывая то, что строчки могут быть довольно длинными. На рис. 11.4 показано, как это странно выглядит:

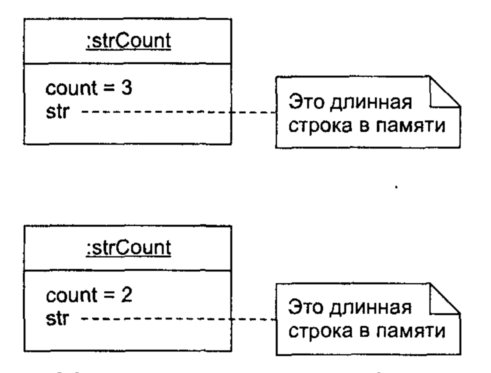


Рис. 11.4. Объектная диаграмма UML: дублирование строк

Вместо того чтобы держать в каждом объекте класса String символьную строку, можно занести в объект только лишь *указатель* на строку! Теперь, присваивая значения одного объекта другому, мы копируем только указатель из одного объекта в другой. Оба указателя, разумеется, будут указывать на одну и ту же строку. Да, это действительно намного лучше, ведь мы храним в памяти только один экземпляр каждой строки. На рис. 11.5 это показано.

Конечно, используя такую систему, необходимо внимательно следить за удалением объектов класса String. Если деструктор этого класса использует delete для освобождения памяти, занятой символьной строкой, и если имеются несколько указателей на эту строку, объекты так и останутся с висящими указателями, указывающими туда, где строки уже давно нет.

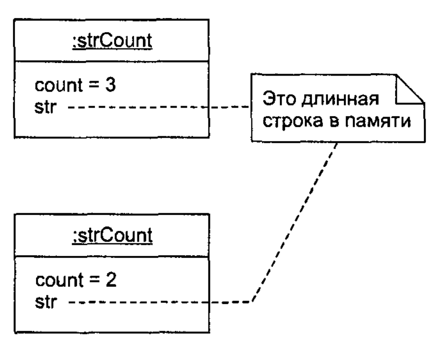


Рис. 11.5. Объектная диаграмма UML: дублирование указателей на строку

Выходит, что для того, чтобы использовать указатели на строки в объектах класса String, необходимо следить за тем, сколько именно объектов указывают на конкретную строку. Тем самым мы можем избежать использования delete по отношению к строкам до тех пор, пока не будет удален последний объект, указывающий на данную строку. Наш следующий пример, STRIMEM, предназначен как раз для демонстрации этого.

Класс-счетчик строк

Допустим, имеется несколько объектов класса String, указывающих на определенную строку, и мы хотим научить программу считать, сколько именно объектов на нее указывают. Где нам хранить этот счетчик?

Было бы слишком обременительно для каждого объекта класса String считать, сколько его коллег указывают на данную строку, поэтому мы не будем вводить счетчик в состав переменных класса String. Может быть, использовать статическую переменную? А ведь это мысль! Мы могли бы создать статический массив, который хранил бы список адресов строк и их порядковые номера. Да, но, с другой стороны, накладные расходы слишком велики. Рациональней будет создать новый особый класс для подсчета строк и указателей. Каждый объект такого класса, назовем его strCount, будет содержать счетчик и сам указатель на строчку. В каждый объект класса String поместим указатель на соответствующий объект класса strCount. Схема такой системы показана на рис. 11.6.

Для того чтобы убедиться в нормальном доступе объектов String к объектам strCount, сделаем String дружественным по отношению к strCount. Кроме того, нам хотелось бы достоверно знать, что класс strCount используется только классом String. Чтобы запретить несанкционированный доступ к каким-либо его функциям, сделаем все методы strCount скрытыми. Поскольку String является дружественным, на него это ограничение не распространяется. Приводим листинг программы STRIMEM.

Листинг 11.18. Программа STRIMEM

```
// strimem.cpp
// Класс String с экономией памяти
// Перегружаемая операция присваивания и конструктор        // копирования
#include <iostream>
#include <cstring>                 // для strcpy() и т. д.
using namespace std;
```

```
////////////////////////////////////////////////////////////
class strCount                     // Класс-счетчик уникальных строк
  {                      private:
    int count;                     // собственно счетчик
    char* str;                     // указатель на строку
    friend class String;           // сделаем себя доступными
    //методы скрыты
//----------------------------------------------------------
    strCount(char* s)              // конструктор с одним аргументом
      {
      int length = strlen(s);      // длина строкового
                                   // аргумента
      str = new char[length+1];    // занять память
                                   // под строку
      strcpy(str, s);              // копировать в нее аргументы
      count=1;                     // считать с единицы
      }
//----------------------------------------------------------
    ~strCount()                    // деструктор
      { delete[] str; }            // удалить строку
  };
////////////////////////////////////////////////////////////
class String                       // класс String
  {
  private:
    strCount* psc;                 // указатель на strCount
  public:
    String()                       // конструктор без аргументов
      { psc = new strCount("NULL"); }
//----------------------------------------------------------
    String(char* s)                // конструктор с одним аргументом
      { psc = new strCount(s); }
//----------------------------------------------------------
    String(String& S)              // конструктор копирования
      {
      psc = S.psc;
      (psc->count)++;
      }
//----------------------------------------------------------
    ~String()                      // деструктор
      {
      if(psc->count==1)            // если последний
                                   // пользователь,
        delete psc;                // удалить strCount
      else                         // иначе
        (psc->count)--;            // уменьшить счетчик
      }
//----------------------------------------------------------
    void display()                 // вывод String
      {
      cout << psc->str;            // вывести строку
      cout << " (addr=" << psc << ")";  // вывести адрес
      }
//----------------------------------------------------------
    void operator = (String& S) // присвоение String
      {
      if(psc->count==1)            // если последний
                                   // пользователь,
```

продолжение ⇨

Листинг 11.18 (*продолжение*)

```
        delete psc;             // удалить strCount
      else                      // иначе
        (psc->count)--;         // уменьшить счетчик
      psc = S.psc;              //использовать strCount
                                //аргумента
      (psc->count)++;           //увеличить счетчик
      }
  };
///////////////////////////////////////////////////////////
int main()
  {
  String s3 = "Муха по полю пошла, муха денежку нашла";
  cout << "\ns3="; s3.display(); //вывести s3

  String s1;                     //определить объект String
  s1 = s3;                       //присвоить его другому объекту
  cout << "\ns1="; s1.display(); //вывести его

  String s2(s3);                 //инициализация
  cout << "\ns2="; s2.display(); //вывести
                                 //инициализированное
  cout << endl;
  return 0;
  }
```

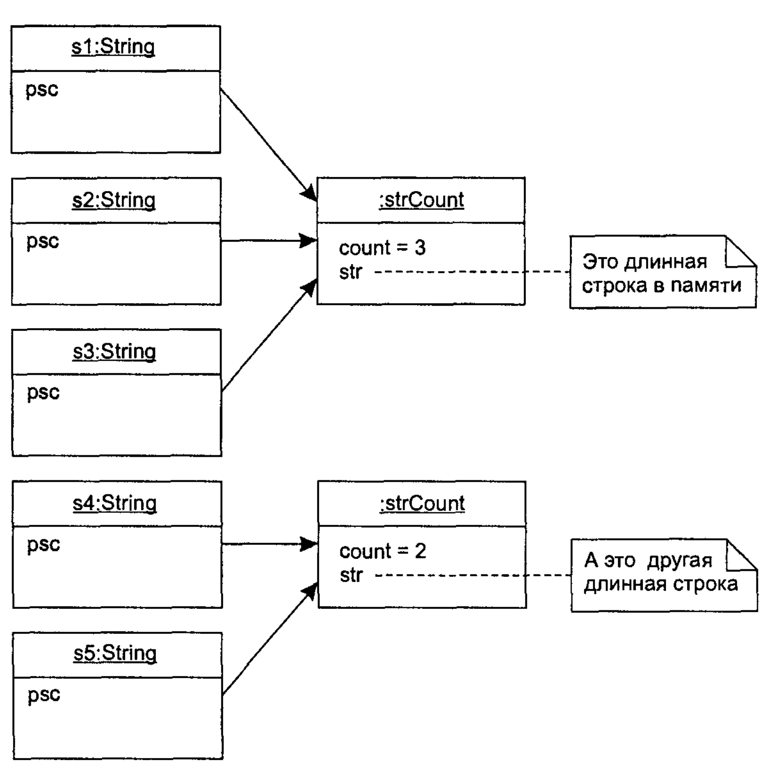


Рис. 11.6. Объекты классов String и strCount

В части main() данной программы мы определяем объект класса String, s3, содержащий строку из известного детского стишка: «Муха по полю пошла, муха денежку нашла». Затем определяем еще один объект, s1, и присваиваем ему значение s3. Определяем s2 и инициализируем его с помощью s3. Полагание s1 равным s3 запускает перегружаемую операцию присваивания, а инициализация s2 с помощью s3 запускает перегружаемую операцию копирования. Выводим все три строки, а также адрес объекта класса strCount, на который ссылается указатель каждого объекта. Мы это делаем для того, чтобы показать, что все строки в действительности представлены одной и той же строкой. Ниже — результат работы программы STRIMEM:

```
s3= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
s2= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
s1= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
```

Остальные обязанности класса String мы поделили между классами String и strCount. Посмотрим, что каждый из них делает.

Класс strCount

Этот класс содержит указатель на реальную строку и считает, сколько объектов класса String на нее указывают. Его единственный конструктор рассматривает указатель на строку в качестве аргумента и выделяет для нее область памяти. Он копирует строку в эту область и устанавливает счетчик в единицу, так как только один объект String указывает на строку сразу после ее создания. Деструктор класса strCount освобождает память, занятую строкой. (Мы используем delete[] с квадратными скобками, так как строка — это массив.)

Класс String

У класса String есть три конструктора. При создании новой строки генерируется новый объект strCount для ее хранения, а указатель psc хранит ссылку на этот объект. Если копируется уже существующий объект String, указатель psc продолжает ссылаться на старый объект strCount, а счетчик в этом объекте увеличивается.

Перегружаемая операция присваивания должна наравне с деструктором удалять старый объект strCount, на который указывает psc, если счетчик равен единице. (Здесь нам квадратные скобки после delete уже не требуются, так как мы удаляем всего лишь единственный объект strCount.) Но почему вдруг оператор присваивания должен заботиться об удалениях? Все дело в том, что объект String, стоящий в выражении слева от знака равенства (назовем его s1), указывал на некоторый объект strCount (назовем его oldStrCnt). После присваивания s1 будет указывать на объект, находящийся справа от знака равенства. А если больше не существует объектов String, указывающих на oldStrCnt, он должен быть удален. Если же еще остались объекты, ссылающиеся на него, его счетчик просто должен быть уменьшен. Рисунок 11.7 показывает действия перегружаемой операции присваивания, а рис. 11.8 показывает конструктор копирования.

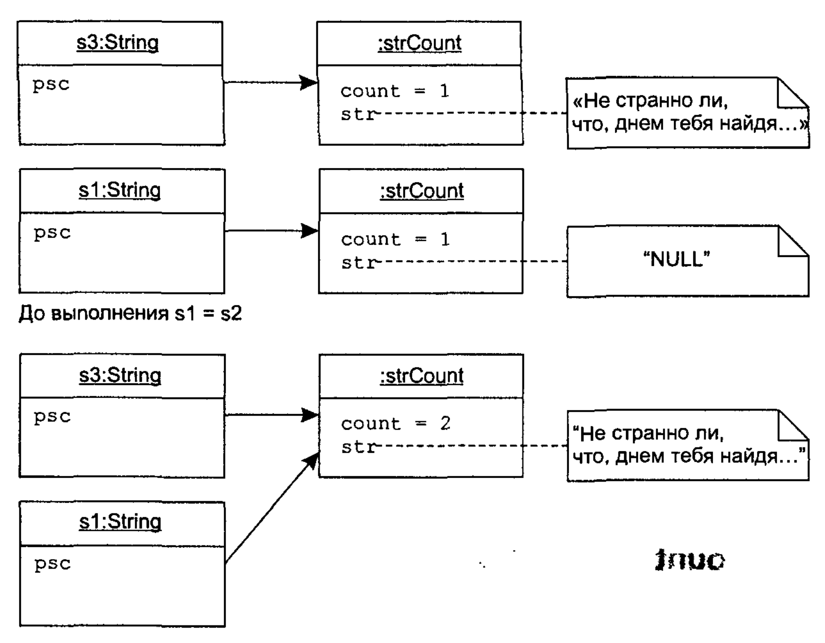


Рис. 11.7. Оператор присваивания в STRIMEM

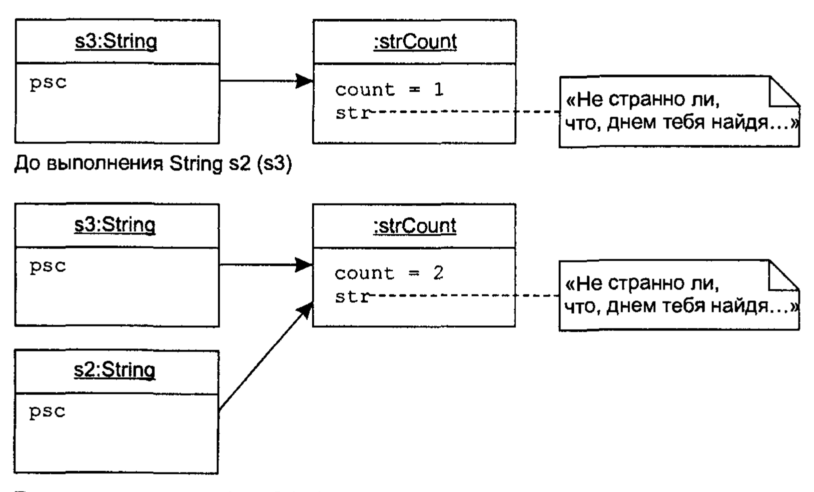


Рис. 11.8. Конструктор копирования в STRIMEM

Указатель this

Методы каждого объекта имеют доступ к некому волшебному указателю под названием this, который ссылается на сам объект. Таким образом, любой метод может узнать адрес, образно говоря, дома, в котором он прописан, то есть адрес сво-

его родного объекта. В следующем примере мы покажем механизм работы с этим загадочным указателем.

Листинг 11.19. Программа WHERE

```
// where.cpp
// указатель this
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class where
  {
  private:
    char charray[10];  //массив из 10 байтов
  public:
    void reveal()
      { cout << "\nМой адрес - не дом и не улица, мой адрес - " << this; }
                       // вывести адрес обьекта
  };
////////////////////////////////////////////////////////////
int main()
  {
  where w1, w2, w3;    //создать три объекта
  w1.reveal();         //посмотреть, где они находятся
  w2.reveal();
  w3.reveal();
  cout << endl;
  return 0;
  }
```

В функции main() данной программы создаются три объекта типа where. Затем печатаются их адреса с помощью метода reveal(). Этот метод просто выводит значение указателя this. Вот как это выглядит на экране:

```
Мой адрес - не дом и не улица, мой адрес - 0x8f4effec
Мой адрес - не дом и не улица, мой адрес - 0x8f4effe2
Мой адрес - не дом и не улица, мой адрес - 0x8f4effd8
```

Так как данные объектов хранятся в массивах, размер каждого из которых равен 10 байтов, то объекты в памяти отделены друг от друга десятью байтами (EC–E2=E2–D8=10 (дес)). Впрочем, некоторые компиляторы могут творчески подойти к расположению объектов и выделить под каждый чуть больше, чем 10 байтов.

Доступ к компонентным данным через указатель this

Когда вы вызываете какой-либо метод, значением указателя this становится адрес объекта, для которого этот метод вызван. Указатель this может интерпретироваться, как любой другой указатель на объект, соответственно, его можно использовать для получения доступа к данным объекта, на который он ссылается. Это продемонстрировано в программе DOTHIS.

Листинг 11.20. Программа DOTHIS

```
// dothis.cpp
// как указатель this ссылается на данные
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class what
  {
  private:
    int alpha;
  public:
    void tester()
      {
      this->alpha = 11;    //то же,что alpha = 11;
      cout << this->alpha; //то же, что cout <<
                           //alpha;
      }
  };
///////////////////////////////////////////////////////////
int main()
  {
  what w;
  w.tester();
  cout << endl;
  return 0;
  }
```

Эта программа просто выводит на экран значение 11. Обратите внимание, каким образом метод tester() получает доступ к переменной alpha:

```
this -> alpha
```

Это точно то же самое, что прямая ссылка на alpha. Такой синтаксис обрабатывается без каких-либо проблем, но в нем на самом деле никакого иного смысла нет, кроме единственного — показать, что указатель this действительно ссылается на объект.

Использование this для возврата значений

Более практичным применением указателя this является возврат значений из методов и перегружаемых операций в вызывающую программу.

Вспомните, что в программе ASSIGN мы никак не могли решить проблему возврата объекта из функции по ссылке, потому что объект был локальный по отношению к возвращающей функции и, следовательно, уничтожался во время выхода из функции. Нам необходим был более долговечный объект, чтобы возвращать результат по ссылке. Объект, чьим методом является данная функция, прочнее, нежели его собственные методы. Методы объекта создаются и уничтожаются при каждом вызове, в то время как сам объект может быть удален только извне (например, по delete). Таким образом, возврат по ссылке результата работы функции, являющейся методом данного объекта, является более профессиональным решением, чем возврат объекта, созданного данным методом. Объясним чуть более простыми словами: будет лучше, если в объект включен

метод, потому что в таком случае этот объект и возвращается в результате работы своего метода в вызывающую программу. Раньше у нас было наоборот — в методе содержался объект, который возвращался в вызывающую программу, но в результате того, что локальный объект при выходе из функции подлежит удалению, мы не могли передать его по ссылке. Новый подход очень легко осуществляется с помощью указателя this.

Ниже приведен листинг программы ASSIGN2, где функция operator=() возвращает по ссылке тот объект, в который она включена.

Листинг 11.21. Программа ASSIGN2

```
 assign2.cpp
  возврат содержимого указателя this
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class alpha
  {
  private:
    int data;
  public:
    alpha()                     // конструктор без аргументов
      { }
    alpha(int d)                // конструктор с одним аргументом
      { data = d; }
    void display()              // вывести данные
      { cout << data; }
    alpha& operator = (alpha& a) // перегружаемая операция =
      {
      data = a.data;            // не делается автоматически
      cout << "\nЗапущен оператор присваивания";
      return *this;             // вернуть копию this alpha
      }
  };
///////////////////////////////////////////////////////////
int main()
  {
  alpha a1(37);
  alpha a2, a3;

  a3 = a2 = a1;                 // перегружаемый =, дважды
  cout << "\na2="; a2.display(); // вывести a2
  cout << "\na3="; a3.display(); // вывести a3
  cout << endl;
  return 0;
  }
```

В этой программе можно использовать описатель

```
alpha& operator = (alpha& a)
```

использующий возврат по ссылке, вместо

```
alpha operator = (alpha& a)
```

возвращающего результат по значению. Последним выражением функции является

```
return *this;
```

Так как this является указателем на объект, чей метод выполняется, то *this — это и есть сам объект, и вышеприведенное выражение возвращает его по ссылке. Вот результат работы программы ASSIGN2:

```
Запущен оператор присваивания
Запущен оператор присваивания
a2=37
a2=37
```

Всякий раз, когда в нашей программе встречается знак равенства

```
a3 = a2 = a1
```

запускается перегружаемая функция operator=(), которая выводит соответствующее сообщение. Все три объекта у нас имеют одно и то же значение.

Обычно описанный выше метод с возвратом результата работы перегружаемого operator=() по ссылке используется для того, чтобы избежать создания никому не нужных дополнительных копий объекта.

Исправленная программа STRIMEM

Используя указатель this, можно переделать функцию operator=() в программе STRIMEM и заставить ее возвращать результат по ссылке. Таким способом можно осуществить множественные присваивания для класса String, например

```
s1 = s2 = s3
```

В то же время мы избежим и создания ложных объектов в виде копий для возврата результата, как было, когда этот возврат осуществлялся по значению. Ниже представлен листинг программы STRIMEM2.

Листинг 11.22. Программа STRIMEM2

```
// strimem2.cpp
// Класс String с экономией памяти
// Перегружаемая операция присваивания и указатель this
#include <iostream>
#include <cstring> //для strcpy() и т. д.
using namespace std;
////////////////////////////////////////////////////////////////
class strCount //Класс-счетчик уникальных строк
{private:
int count;//собственно счетчик
char* str;//указатель на строку
friend class String; //сделаем себя доступными
//методы скрытые
strCount(char* s)//конструктор с одним аргументом
{
int length = strlen(s); //длина строкового //аргумента
str = new char[length+1]; //занять память   //под строку
strcpy(str, s);//копировать в нее аргументы
count=1; //считать с единицы
}
//--------------------------------------------------------
~strCount()//деструктор
{ delete[] str; }//удалить строку
};
```

```
///////////////////////////////////////////////////////////
class String //класс String
{
private:
strCount* psc: //указатель на strCount
public:
String()  //конструктор без аргументов
{ psc = new strCount("NULL"): }
//---------------------------------------------------------
String(char* s) //конструктор с одним аргументом
{ psc = new strCount(s): }
//---------------------------------------------------------
String(String& S)  //конструктор копирования
{
cout << "\nКОНСТРУКТОР КОПИРОВАНИЯ":
psc = S.psc:
(psc->count)++:
}
//---------------------------------------------------------
~String() //деструктор
{
if(psc->count==1)  //если последний   //пользователь.
delete psc: //удалить strCount
else//иначе
(psc->count)--://уменьшить счетчик
}
//---------------------------------------------------------
void display()  //вывод String
{
cout << psc->str: //вывести строку
cout << " (addr=" << psc << ")":  //вывести      //адрес
}
//---------------------------------------------------------
String& operator = (String& S)//присвоение String
{
cout << "\nПРИСВАИВАНИЕ":
if(psc->count==1)  //если последний       //пользователь.
delete psc://удалить strCount
else//иначе
(psc->count)--://уменьшить счетчик
psc = S.psc: //использовать strCount  //аргумента
(psc->count)++://увеличить счетчик
return *this://вернуть этот объект
}
}:
///////////////////////////////////////////////////////////
int main()
{
String s3 = "Муха по полю пошла, муха денежку нашла":
cout << "\ns3=": s3.display()://вывести s3

String s1,s2:    //определить объекты String
s1 = s2 = s3://присваивания
//вывести их
cout << "\ns1=": s1.display():
cout << "\ns2=": s2.display():
cout << endl://ожидать нажатия клавиши
return 0:
}
```

Теперь описателем оператора присваивания является

```
String& operator = (String& S)//возврат по ссылке
```

и, как в программе ASSIGN2, функция возвращает указатель на this. Результат работы программы:

```
s3=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
ПРИСВАИВАНИЕ
ПРИСВАИВАНИЕ
s2=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
s1=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
```

Результат работы программы показывает, что в соответствии с выражением присваивания все три объекта класса String указывают на один и тот же объект класса strCount.

Следует отметить, что указатель this нельзя использовать в статических методах, так как они не ассоциированы с конкретным объектом.

Остерегайтесь неправильных присваиваний

Один из основополагающих так называемых «законов Мёрфи» говорит о том, что все, что можно случайно сделать не так, обязательно будет кем-нибудь сделано. Это абсолютная правда в отношении программирования, поэтому будьте уверены: если вы переопределили оператор присваивания, кто-нибудь станет его использовать для присваивания объектов самим себе. Вот так:

```
alpha = alpha;
```

Перегружаемый = должен быть морально готов обработать такую ситуацию. В противном случае не ждите ничего хорошего. Например, в секции main() программы STRIMEM2, если вы положите объект String равным самому себе, программа зависнет (несмотря на то что другие объекты без проблем используют один и тот же объект strCount). Проблема заключается в том, что согласно коду перегружаемой операции присваивания последний удаляет объект strCount, если считает, что вызываемый объект является единственным, использующим strCount. Присваивание объекта самому себе убедит оператор в этом прискорбном факте, хотя на самом деле никто и не собирался удалять никакие объекты.

Чтобы привести все в порядок, нужно ввести проверку на присваивание самому себе в начале работы каждого перегружаемого =. В большинстве случаев это делается путем сравнения адресов объектов, стоящих справа и слева от знака равенства. Понятно, что если адреса равны, значит, осуществляется присваивание объекта самому себе и необходимо немедленно вернуться из этой функции. (Если кто-то не понял, в чем проблема, поясняем, что не нужно присваивать объекты самим себе — они и так уже равны.) Например, в STRIMEM2 с этой целью можно вставить следующие две строчки

```
if(this == &S)
   return *this;
```

в начало функции operator=(). Это должно решить возникшую проблему.

Динамическая информация о типах

Существует возможность узнавать информацию о классе объекта и даже изменять этот класс прямо во время выполнения программы. Мы вкратце рассмотрим два механизма, которые служат для этого: операция dynamic_cast и оператор typeid. Их можно рассматривать как дополнительные средства языка, но когда-нибудь они могут и пригодиться.

Эти возможности используются, когда базовый класс имеет ряд порожденных классов, созданных порой довольно непростым путем. Для того чтобы использовать динамический подход, базовый класс обязан быть полиморфным, то есть он должен содержать по крайней мере одну виртуальную функцию.

Чтобы заработали dynamic_cast и typeid, компилятор должен активизировать механизм, который позволяет определять и изменять тип объекта во время выполнения программы — RTTI (Run-Time Type Information). В системе Borland C++ Builder этот механизм включается по умолчанию, а в Microsoft Visual C++ нужно подключать его вручную. Также необходимо включить в программу заголовочный файл TYPEINFO.

Проверка типа класса с помощью dynamic_cast

Допустим, некая внешняя программа передает в вашу программу объект (так любит делать операционная система с функцией обратного вызова). Предполагается, что объект принадлежит какому-то известному классу, но вам хочется проверить это. Как это сделать? Как определить класс объекта? Оператор dynamic_cast позволяет это сделать, предполагая, что классы, чьи объекты вы хотите проверить, все вышли из гоголевской «Шинели», то есть имеют общий базовый класс. Этот подход демонстрируется в программе DYNCAST1:

Листинг 11.23. Программа DYNCAST1

```
//dyncast1.cpp
//динамическое приведение типов для проверки типа объекта
//RTTI должен быть включен
#include <iostream>
#include <typeinfo>          //для dynamic_cast
using namespace std;
////////////////////////////////////////////////////////////
class Base
  {
  virtual void vertFunc()    //для dynamic cast
    { }
  };
class Derv1 : public Base
  { };
class Derv2 : public Base
  { };
```

продолжение ⇨

Листинг 11.23 (*продолжение*)

```
///////////////////////////////////////////////////////////
//проверка, указывает ли pUnknown на Derv1
bool isDerv1(Base* pUnknown) //неизвестный подкласс базового
  {
  Derv1* pDerv1;
  if( pDerv1 = dynamic_cast<Derv1*>(pUnknown) )
    return true;
  else
    return false;
  }
//---------------------------------------------------------
int main()
  {
  Derv1* d1 = new Derv1;
  Derv2* d2 = new Derv2;

  if( isDerv1(d1) )
    cout << "d1 - компонент класса Derv1\n";
  else
    cout << "d1 - не компонент класса Derv1\n";

  if( isDerv1(d2) )
    cout << "d2 - компонент класса Derv1\n";
  else
    cout << "d2 - не компонент класса Derv1\n";
  return 0;
  }
```

У нас есть базовый класс Base и два порожденных — Derv1 и Derv2. Есть также функция isDerv1(), возвращающая значение «истина», если указатель, переданный в качестве аргумента, указывает на объект класса Derv1. Наш аргумент принадлежит классу Base, поэтому объекты могут быть либо класса Derv1, либо Derv2. Оператор dynamic_cast пытается привести указатель неизвестного типа pUnknown к типу Derv1. Если результат ненулевой, значит, pUnknown указывал на объект Derv1. В противном случае он указывал на что-то другое.

Изменение типов указателей с помощью dynamic_cast

Оператор dynamic_cast позволяет осуществлять приведение типов «вверх» и «вниз» по генеалогическому дереву классов. Но его возможности имеют некоторые ограничения. Покажем в программе DYNCAST2 порядок работы с этим оператором.

Листинг 11.24. Программа DYNCAST2

```
//dyncast2.cpp
//Тестирование динамического приведения типов
//RTTI должен быть включен
#include <iostream>
#include <typeinfo>          //для dynamic_cast
using namespace std;
///////////////////////////////////////////////////////////
class Base
```

```
  {
  protected:
    int ba;
  public:
    Base() : ba(0)
      {  }
    Base(int b) : ba(b)
      {  }
    virtual void vertFunc()  //для нужд dynamic_cast
      {  }
    void show()
      { cout << "Base: ba=" << ba << endl; }
  };
////////////////////////////////////////////////////////////////
class Derv : public Base
  {
  private:
    int da;
  public:
    Derv(int b, int d) : da(d)
      { ba = b; }
    void show()
      { cout << "Derv: ba=" << ba << ", da=" << da << endl; }
  };
////////////////////////////////////////////////////////////////
int main()
  {
  Base* pBase = new Base(10);     //указатель на Base
  Derv* pDerv = new Derv(21, 22); //указатель на Derv

  //приведение к базовому типу: восхождение  по дереву -
  //указатель поставлен на подобъект Base класса Derv
  pBase = dynamic_cast<Base*>(pDerv);
  pBase->show();                  //"Base: ba=21"

  pBase = new Derv(31, 32);       //обычное нисходящее
   //приведение типов -- pBase должен указывать на Derv)
  pDerv = dynamic_cast<Derv*>(pBase);
  pDerv->show();                  //"Derv: ba=31, da=32"
  return 0;
  }
```

В этом примере у нас есть базовый и порожденный классы. В каждый из них мы ввели элемент данных для лучшей демонстрации эффекта динамического приведения типов.

При восходящем приведении типов производится попытка изменить объект порожденного класса на объект базового класса. Все, что вы при этом можете получить, — это только унаследованная из базового класса часть объекта порожденного. В нашем примере мы создали объект класса Derv. Ему в наследство от базового класса досталась переменная ba, которая, будучи в базовом классе, имела значение 21, а в порожденном классе имела значение 22. После приведения типов указатель pBase стал ссылаться на базовую часть объекта класса Derv1, поэтому на экране мы видим: Base:ba=21. Восходящее приведение типов целесообразно использовать, когда вам нужно получить базовую составляющую объекта.

При нисходящем приведении типов мы переносим ссылку на объект порожденного класса из указателя базового класса в указатель порожденного.

Оператор typeid

Иногда требуется более полная информация об объекте, нежели простая верификация его классовой принадлежности. Можно узнать, например, имя типа неопознанного объекта, используя оператор typeid. Рассмотрим в связи с этим программу TYPEID.

Листинг 11.25. Программа TYPEID

```
// typeid.cpp
// демонстрация функции typeid()
// RTTI должен быть активизирован
#include <iostream>
#include <typeinfo>         // для typeid()
using namespace std;
////////////////////////////////////////////////////////////////
class Base
  {
  virtual void virtFunc()  // для нужд typeid
    {  }
  };
class Derv1 : public Base
  { };
class Derv2 : public Base
  { };
////////////////////////////////////////////////////////////////
void displayName(Base* pB)
  {
  cout << "указатель на обьект класса "; // вывести имя класса
  cout << typeid(*pB).name() << endl;    //на который
                                         //указывает pB
  }
//-------------------------------------------------------------
int main()
  {
  Base* pBase = new Derv1;
  displayName(pBase);  //"указатель на обьект класса Derv1"
  pBase = new Derv2;
  displayName(pBase);  //" указатель на обьект класса Derv2"
  return 0;
  }
```

В этом примере функция displayName() отображает имя класса переданного ей объекта. Для этого она использует переменную name класса type_info наряду с оператором typeid. В main() мы передаем этой функции два объекта классов Derv1 и Derv2 соответственно. На экране видим следующее:

```
указатель на обьект класса Derv1
указатель на обьект класса Derv2
```

Используя typeid, кроме имени можно получить и некоторую другую информацию о классе. Например, используя перегружаемую операцию ==, можно

осуществить проверку на равенство классов. Пример этого мы покажем в программе EMPL_IO в главе 12 «Потоки и файлы». Несмотря на то что в этом разделе в примерах использовались указатели, dynamic_cast и typeid работают одинаково хорошо со ссылками.

Резюме

Виртуальные функции позволяют решать прямо в процессе выполнения программы, какую именно функцию вызывать. Без их использования это решение принимается на этапе компиляции программы. Виртуальные функции дают большую гибкость при выполнении одинаковых действий над разнородными объектами. В частности, они разрешают использование функций, вызванных из массива указателей на базовый класс, который на самом деле содержит указатели (или ссылки) на множество порожденных классов. Это пример *полиморфизма*. Обычно функция объявляется виртуальной в базовом классе, а другие функции с тем же именем объявляются в порожденных классах.

Использование по крайней мере одной чистой виртуальной функции делает весь класс, содержащий ее, *абстрактным*. Это означает, что с помощью этого класса нельзя создавать никакие объекты.

Дружественная функция имеет доступ к скрытым данным класса, по отношению к которому она объявлена таковой. Это бывает полезно, когда одна функция должна иметь доступ к двум и более не связанным между собой классам и когда перегружаемая операция должна, со своей стороны, использовать данные «чужого» класса, методом которого она не является. Дружественность также используется для упрощения записи функций.

Статической называется функция, работающая с классом в целом, а не над отдельными его объектами. В частности, она может обрабатывать статические переменные. Она может быть вызвана с использованием имени класса и оператора явного задания функции.

Оператор присваивания = может быть перегружен. Это необходимо, когда он должен выполнять более сложную работу, нежели простое копирование содержимого одного объекта в другой. Конструктор копирования, создающий копии объектов во время инициализации и во время передачи аргументов и возврата результата по значению, также может быть перегружен. Это необходимо, когда он должен выполнять более сложную работу, чем простое копирование объекта.

Указатель this может использоваться в функции для указания на объект, чьим методом он является. Он пригодится, если в качестве результата возвращается объект, чьим методом является данная функция.

Оператор dynamic_cast умеет многое. С его помощью можно определить, на объект какого типа ссылается указатель, можно в некоторых ситуациях изменять тип указателя. Оператор typeid позволяет узнать кое-что о классе объекта, например его название.

Объектная диаграмма UML показывает отношения внутри группы объектов в определенный момент работы программы.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Виртуальные функции позволяют:

   а) создавать массивы типа «указатель на базовый класс», которые могут содержать указатели на производные классы;

   б) создавать функции, к которым никогда не будет доступа;

   в) группировать объекты разных классов так, чтобы они все были доступны с помощью одного и того же вызова функции;

   г) использовать один и тот же вызов функции для выполнения методов объектов, принадлежащих разным классам.

2. Истинно ли утверждение о том, что указатель на базовый класс может ссылаться на объекты порожденного класса?

3. Пусть указатель p ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется невиртуальный метод ding(). Тогда выражение p->ding(); поставит на выполнение версию функции ding() из ____________ класса.

4. Напишите описатель для виртуальной функции dang(), возвращающей результат типа void и имеющей аргумент типа int.

5. Принятие решения о том, какая именно функция будет выполняться по конкретному вызову функции, называется ____________________.

6. Пусть указатель p ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется виртуальный метод ding(). Тогда выражение p->ding(); поставит на выполнение версию функции ding() из ____________ класса.

7. Напишите описатель чистой виртуальной функции aragorn, не возвращающей значений и не имеющей аргументов.

8. Чистая виртуальная функция — это виртуальная функция, которая:

   а) делает свой класс абстрактным;

   б) не возвращает результата;

   в) используется в базовом классе;

   г) не имеет аргументов.

9. Напишите определение массива parr, содержащего 10 указателей на объекты класса dong.

10. Абстрактный класс используется, когда:

    а) не планируется создавать порожденные классы;

    б) есть несколько связей между двумя порожденными классами;

    в) с его помощью запрещено создавать какие-либо объекты;

    г) вы хотите отложить объявление класса.

11. Истинно ли утверждение о том, что дружественная функция имеет доступ к скрытым данным класса, даже не являясь его методом?
12. Дружественная функция может быть использована для того, чтобы:
    а) разные классы могли пользоваться одними аргументами;
    б) разрешать доступ к классам, исходный код которых недоступен;
    в) разрешать доступ к несвязанному классу;
    г) увеличить многосторонность применения перегружаемой операции.
13. Напишите описатель дружественной функции harry(), возвращающей результат типа void и имеющей один аргумент класса george.
14. Ключевое слово friend появляется в:
    а) классе, разрешающем доступ к другому классу;
    б) классе, требующем доступа к другому классу;
    в) разделе скрытых компонентов класса;
    г) разделе общедоступных компонентов класса.
15. Напишите описатель, который в том классе, где он появится, сделает каждый компонент класса harry дружественной функцией.
16. Статическая функция:
    а) должна вызываться, когда объект уничтожается;
    б) сильно связана с индивидуальным объектом класса;
    в) может быть вызвана с использованием имени класса и имени функции;
    г) используется для создания простого объекта.
17. Объясните, что делает по умолчанию оператор присваивания = в применении к объектам.
18. Напишите описатель перегружаемой операции присваивания для класса zeta.
19. Оператор присваивания может быть перегружен с целью:
    а) хранения информации о количестве одинаковых объектов;
    б) присваивания идентификационного номера каждому объекту;
    в) проверки того, что все компонентные данные скопировались без ошибок;
    г) уведомления о том, что имело место присваивание.
20. Истинно ли утверждение о том, что пользователь всегда должен определять операцию для конструктора копирования?
21. Операции, выполняемые оператором присваивания и конструктором копирования:
    а) похожи, за исключением того, что конструктор копирования создает новый объект;
    б) похожи, за исключением того, что оператор присваивания копирует компонентные данные;

в) различны, за исключением того, что оба создают новый объект;

г) различны, за исключением того, что оба копируют компонентные данные.

22. Напишите описатель конструктора копирования для класса Bertha.
23. Истинно ли утверждение о том, что конструктор копирования может быть переопределен с целью копирования только части данных объекта?
24. Продолжительность жизни переменной, которая является:

а) локальной по отношению к методу, совпадает с продолжительностью жизни функции;

б) глобальной, совпадает с продолжительностью жизни класса;

в) нестатическим компонентным данным объекта, совпадает с продолжительностью жизни объекта;

г) статической внутри метода, совпадает с продолжительностью жизни этой функции.

25. Истинно ли утверждение о том, что возврат по значению локальной переменной метода не создает никаких проблем?
26. Объясните разницу в выполнении следующих двух выражений:

```
person p1(p0);
person p1 = p0;
```

27. Конструктор копирования запускается, когда:

а) функция возвращается по значению;

б) аргумент передается по значению;

в) функция возвращается по ссылке;

г) аргумент передается по ссылке.

28. На что ссылается указатель this?
29. Если в заданном классе переменная da является компонентными данными, присвоит ли выражение this.da=37; значение 37 переменной da?
30. Напишите выражение, с помощью которого функция может возвращать весь объект, методом которого она является, без создания временных объектов.
31. Прямоугольник в объектной диаграмме означает:

а) общую группу объектов;

б) класс;

в) экземпляр класса;

г) все объекты класса.

32. Линии между объектами в объектной диаграмме UML называются _____.
33. Истинно ли утверждение о том, что объект A может быть связанным с объектом B только в данный момент и более ни в какой иной?

34. Объектные диаграммы показывают, какие объекты:
   а) существуют в данный момент;
   б) взаимодействуют в данный момент;
   в) принимают участие в каких-то действиях в данный момент;
   г) имеют операции (методы), вызывающие объекты других классов.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Пусть имеется та же издательская компания, которая описана в упражнении 1 главы 9, которая продает и книги, и аудио версии печатной продукции. Как и в том упражнении, создайте класс publication, хранящий название (фактически, строку) и цену (типа float) публикации. Создайте два порожденных класса: book, в котором происходит изменение счетчика страниц (типа int), и tape, в котором происходит изменение счетчика записанных на кассету минут. Каждый из классов должен иметь метод getdata(), запрашивающий информацию у пользователя, и putdata() для вывода данных на экран.

2. Напишите main(), где создавался бы массив указателей на класс publication. Это очень похоже на то, что мы делали в текущей главе на примере VIRTPERS. В цикле запрашивайте у пользователя данные о конкретной книге или кассете, используйте new для создания нового объекта book или tape. Сопоставляйте указатель в массиве с объектом. Когда пользователь закончит ввод исходных данных, выведите результат для всех введенных книг и кассет, используя цикл for и единственное выражение

```
pubarr[j]->putdata(); для вывода данных о каждом объекте из массива
```

*3. В классе Distance, как показано в примерах FRENGL и FRISQ из этой главы, создайте перегружаемую операцию умножения *, чтобы можно было умножать два расстояния. Сделайте эту функцию дружественной, тогда можно будет использовать выражение типа Wdist=7.5*dist2. Вам понадобится конструктор с одним аргументом для перевода величин из формата чисел с плавающей запятой в формат Distance. Напишите какой-либо main() на свое усмотрение для того, чтобы несколькими способами проверить работу этой перегружаемой операции.

*4. Как уже говорилось, классы можно заставлять вести себя как массивы. Пример CLARRAY показывает один из способов создания такого класса.

Листинг 11.26. Программа CLARRAY

```
// clarray.cpp
// создает класс-массив
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Array                    //моделирует обычный массив C++
  {
```

продолжение ⇨

Листинг 11.26 (*продолжение*)

```
  private:
    int* ptr;                  //указатель на содержимое Array
    int size;                  //размер Array
  public:
    Array(int s)               //конструктор с одним аргументом
      {
      size = s;                //аргумент - размер Array
      ptr = new int[s];        //выделить память под Array
      }
    ~Array()                   //деструктор
      { delete[] ptr; }
    int& operator [] (int j)   //перегружаемая операция
                               //списка индексов
      { return *(ptr+j); }
  };
//////////////////////////////////////////////////////////
int main()
  {
  const int ASIZE = 10;        //размер массива
  Array arr(ASIZE);            //создать массив

  for(int j=0; j<ASIZE; j++)   //заполнить его j^2
    arr[j] = j*j;

  for(j=0; j<ASIZE; j++)       //вывести его содержимое
    cout << arr[j] << ' ';
  cout << endl;
  return 0;
  }
```

Результат работы программы:

```
0 1 4 9 16 25 36 49 64 81
```

Взяв за основу приведенную программу, добавьте перегружаемое присваивание и перегружаемый конструктор копирования к классу Array. Затем добавьте к main() выражение Array arr2(arr1); и arr3=arr1; для проверки того, что перегружаемые операции работают. Конструктор копирования должен создать новый объект Array со своим собственным местом в памяти, выделенным для хранения элементов массива. И конструктор копирования, и оператор присваивания должны копировать содержимое старого объекта класса Array в новый. Что будет, если вы присвоите объект Array одного размера объекту Array другого размера?

5. Взяв за основу программу из упражнения 1 этой главы, добавьте метод типа bool, называющийся isOveersize(), к классам book и tape. Допустим, книга, в которой больше 800 страниц, или кассета со временем проигрывания более 90 минут, будут считаться объектами с превышением размера. К этой функции можно обращаться из main(), а результат ее работы выводить в виде строки «Превышение размера!» для соответствующих книг и кассет. Допустим, объекты классов book и tape должны быть доступны через указатели на них, хранящиеся в массиве типа publication.

Что в этом случае вам нужно добавить в базовый класс publication? Вы можете привести примеры компонентов этого базового класса?

6. Возьмите за основу программу из упражнения 8 главы 8, где было перегружено пять арифметических операций для работы с денежным форматом. Добавьте два оператора, которые не были перегружены в том упражнении:

```
long double * bMoney      //умножать число на деньги
long double / bMoney      //делить число на деньги
```

Эти операции требуют наличия дружественных функций, так как справа от оператора находится объект, а слева — обычное число. Убедитесь, что main() позволяет пользователю ввести две денежные строки и число с плавающей запятой, а затем корректно выполняет все семь арифметических действий с соответствующими парами значений.

7. Как и в предыдущем упражнении, возьмите за основу программу из упражнения 8 главы 8. На этот раз от вас требуется добавить функцию, округляющую значение bMoney до ближайшего доллара:

```
mo2 = round(mo1);
```

Как известно, значения, не превышающие $0.49, округляются вниз, а числа от $0.50 и более округляются вверх. Можно использовать библиотечную функцию modfl(). Она разбивает переменную типа long double на целую и дробную части. Если дробная часть меньше 0.50, функция просто возвращает целую часть числа. В противном случае возвращается увеличенная на 1 целая часть. В main() проверьте работоспособность функции путем передачи в нее последовательно значений, одни из которых меньше $0.49, другие — больше $0.50.

8. Помните программу PARSE из главы 10? Попробуйте доработать ее, чтобы она могла вычислять значения математических выражений с рациональными числами, например типа float, а не только с одноразрядными числами:

```
3.14159 / 2.0 + 75.25 * 3.333 + 6.02
```

Во-первых, нужно развить стек до такой степени, чтобы он мог хранить и операторы (типа char), и числа (типа float). Но как, спрашивается, можно хранить в стеке значения двух разных типов? Ведь стек — это, по сути дела, массив. Надо еще учесть, что типы char и float даже не совпадают по размеру! Даже указатели на разные типы данных (char* и float*) компилятор не позволит хранить в одном массиве, несмотря на то, что они одинакового размера. Единственный способ хранить в массиве два разных типа указателей — сделать эти типы наследниками одного и того же базового класса. При этом базовому классу даже нет нужды иметь какие-то собственные данные, это может быть абстрактный класс, из которого никакие объекты создаваться не будут.

Конструкторы могут хранить значения в порожденных классах обычным способом, но должна иметься специальная чистая виртуальная функция

для того, чтобы извлечь эти значения. Представляем возможный сценарий работы над этим вопросом:

```
class Token                    // Абстрактный базовый класс
  {
  public:
    virtual float getNumber()=0; // чистая виртуальная
                                 // функция
    virtual char getOperator()=0;
  };
class Operator : public Token
  {
  private:
    char oper;             // Операторы +, -, *, /
  public:
    Operator(char);        // конструктор устанавливает значение
    char getOperator();    // получить значение
    float getNumber();     // просто некая функция
  };
class Number : public Token
  {
  private:
    float fnum;            // число
  public:
    Number(float);         // конструктор устанавливает значение
    float getNumber();     // получить значение
    char getOperator();    // просто некая функция
  };
Token* atoken[100];    //содержит типы Operator* и Number*
```

Виртуальные функции базового класса должны быть реализованы во всех порожденных классах, в противном случае классы становятся абстрактными. Таким образом, классу Operand нужна функция getNumber(), несмотря на то, что она фиктивная. Классу Number нужна функция getOperand(), несмотря на то, что она тоже фиктивная.

Поработайте над этим каркасом, сделайте его реально работающей программой, добавив класс Stack, содержащий объекты класса Token, и функцию main(), в которой бы заносились в стек и извлекались из него разные арифметические операторы и числа в формате с плавающей запятой.

9. Внесем некоторое разнообразие в пример HORSE из главы 10, создав класс для внесения в него лошадей экстракласса. Предположим, что любая лошадь, которая на скачках к середине дистанции находится впереди всех, становится практически непобедимой. Относительно класса лошадей создадим порожденный класс comhorse (для конкурентоспособной лошади). Перегрузим функцию horse_tick() в этом классе таким образом, чтобы каждая лошадь могла проверять, является ли она ведущей и нет ли поблизости соперников (скажем, ближе, чем на 0,1 форлонг (1/80 часть мили или 20,1 м.)). Если есть, то ей следует немного ускориться. Может быть, не настолько, чтобы побеждать на всех скачках, но в достаточной мере для того, чтобы оставаться конкурентоспособной.

Как каждая лошадь узнает, где находятся остальные? Моделирующий ее объект должен иметь доступ к области памяти, в которой хранятся дан-

ные о соперниках. В программе HORSE это hArray. Будьте внимательны: вы создаете класс для передовой лошади, он должен быть наследником класса всех лошадей. Поэтому классу comhorse потребуется перегрузить hArray. Вам может потребоваться создать еще один производный класс, comtrack, для отслеживания позиции лошади.

Можно непрерывно проверять, лидирует ли ваша лошадь, и если она впереди всех, но лишь ненамного, следует ее немного ускорить.

10. Упражнение 4 в главе 10 включало в себя добавление к классу linklist перегружаемого деструктора. Допустим, мы заполняем объект этого класса данными, а затем присваиваем один класс целиком другому, используя стандартный оператор присваивания:

```
list2 = list1;
```

Допустим, что впоследствии мы удалим объект класса list1. Можем ли мы все еще использовать list2 для доступа к введенным данным? Увы, нет, так как при удалении list1 все его ссылки были удалены. Единственное, что было известно объекту linklist про удаленный объект, это указатель на него. Но его удалили, указатель в list2 стал недееспособным, и все попытки получить доступ к данным приведут к получению мусора вместо данных, а в худшем случае — к зависанию программы.

Один из способов избежать этих проблем — перегрузить оператор присваивания, чтобы он вместе с объектом копировал бы все его ссылки. Но тогда придется пройти по всей цепочке, поочередно копируя все ссылки. Как отмечалось ранее, следует также перегружать конструктор копирования. Чтобы была возможность удалять объекты linklist в main(), можно создавать их с помощью указателя и new. В таком случае проще будет проверять работу новых операций. Не переживайте, если обнаружите, что в процессе копирования порядок следования данных изменился.

Понятно, что копирование всех данных не является самым эффективным решением проблемы с точки зрения экономии памяти. Сравните этот подход с представленным в примере STRIMEM (глава 10), где использовался только один набор данных для всех объектов, и хранилась информация о том, сколько объектов указывали на эти данные.

11. Выполните изменения в соответствии с упражнением 7, применив их к программе PARSE главы 10. То есть заставьте программу анализировать выражения, содержащие числа в формате с плавающей запятой. Совместите классы, предложенные в упражнении 7, с алгоритмами из PARSE. Вам придется работать с указателями на символы вместо работы с самими символами. Это потребует выражений, подобных следующим:

```
Number* ptrN = new Number(ans);
s.push(ptrN);
and
Operator* ptrO = new Operator(ch);
s.push(ptrO);
```

Глава 12
Потоки и файлы

- ✦ Потоковые классы
- ✦ Ошибки потоков
- ✦ Потоковый ввод/вывод дисковых файлов
- ✦ Указатели файлов
- ✦ Обработка ошибок файлового ввода/вывода
- ✦ Файловый ввод/вывод с помощью методов
- ✦ Перегрузка операций извлечения и вставки
- ✦ Память как поток
- ✦ Аргументы командной строки
- ✦ Вывод на печатающее устройство

Эта глава посвящена потоковым классам в C++. Мы начнем с иерархии, в соответствии с которой организуются эти классы, и сведем вместе знания обо всех их важных особенностях. Наибольшее внимание в главе уделено тому, чтобы показать, как выполняются действия, связанные с файлами, с использованием потоков. Мы покажем, как выполняется чтение и запись файлов различными способами, как обрабатывать ошибки ввода/вывода, как связано ООП с файлами. Затем мы рассмотрим некоторые другие особенности C++, связанные с файлами, включая форматирование текста «в памяти» (файл «в памяти» ведет себя, как дисковый, хотя находится в оперативной памяти), работу с аргументами командной строки, перегрузку операторов вставки и извлечения, посылку данных на принтер.

Потоковые классы

Поток — это общее название, как ни странно, потока данных. В C++ поток представляет собой объект некоторого класса. Именно поэтому вы могли встретить в листингах потоковые объекты cin и cout. Разные потоки предназначены для

представления разных видов данных. Например, класс ifstream олицетворяет собой поток данных от входного дискового файла.

Преимущества потоков

Программисты на С могут удивиться, какие же преимущества дает использование потоковых классов для ввода/вывода вместо традиционных функций С printf(), scanf(), для файлов — fprintf(), fscanf()...

Одним из аргументов в пользу потоков является простота использования. Если вам приходилось когда-нибудь использовать символ управления форматом %d при форматировании вывода с помощью %f в printf(), вы оцените это. Ничего подобного в потоках вы не встретите, ибо каждый объект сам знает, как он должен выглядеть на экране. Это избавляет программиста от одного из основных источников ошибок.

Другой причиной является то, что можно перегружать стандартные операторы и функции вставки (<<) и извлечения (>>) для работы с создаваемыми классами. Это позволяет работать с собственными классами как со стандартными типами, что, опять же, делает программирование проще и избавляет от множества ошибок, не говоря уж об эстетическом удовлетворении.

Но неужели потоковый ввод/вывод так важен при работе в среде с графическим интерфейсом пользователя, такой, как, например, Windows, где непосредственный вывод текста на экран не используется вообще? Неужели в наше время еще нужны потоки С++? Оказывается, да, нужны. Потому что это лучший способ записывать данные в файл, лучший способ организации данных в памяти для последующего использования при вводе/выводе текста в окошках и других элементах графического интерфейса пользователя (GUI).

Иерархия потоковых классов

Потоковые классы имеют довольно сложную иерархическую структуру. На рис. 12.1 показана организация важнейших из них.

Мы уже встречались с использованием некоторых потоковых классов. Операция извлечения >> является методом класса istream, операция вставки << — методом класса ostream. Оба этих класса являются наследниками ios. Объект cout, представляющий собой стандартный выходной поток, который обычно выводит на экран данные, является предопределенным объектом класса ostream_withassign, являющегося наследником класса ostream. Аналогично, cin — объект istream_withassign, наследника istream.

Классы, используемые для вывода данных на экран и ввода с клавиатуры, описаны в заголовочном файле IOSTREAM, который мы всегда подключали в наших примерах из предыдущих глав. Классы, используемые для ввода/вывода файлов, объявлены в файле FSTREAM. На рис. 12.1 видно, в каких заголовочных файлах что хранится. (К сказанному можно добавить, что некоторые манипуляторы описаны в IOMANIP, а некоторые классы для работы с объектами «в памяти» определены в STRSTREAM.) В качестве полезного упражнения можно распечатать эти заголовочные файлы и проследить взаимоотношения различных

классов. Файлы хранятся в директории INCLUDE вашего компилятора. При изучении хранящихся в этих файлах классов можно найти ответы на многие вопросы, касающиеся потоков.

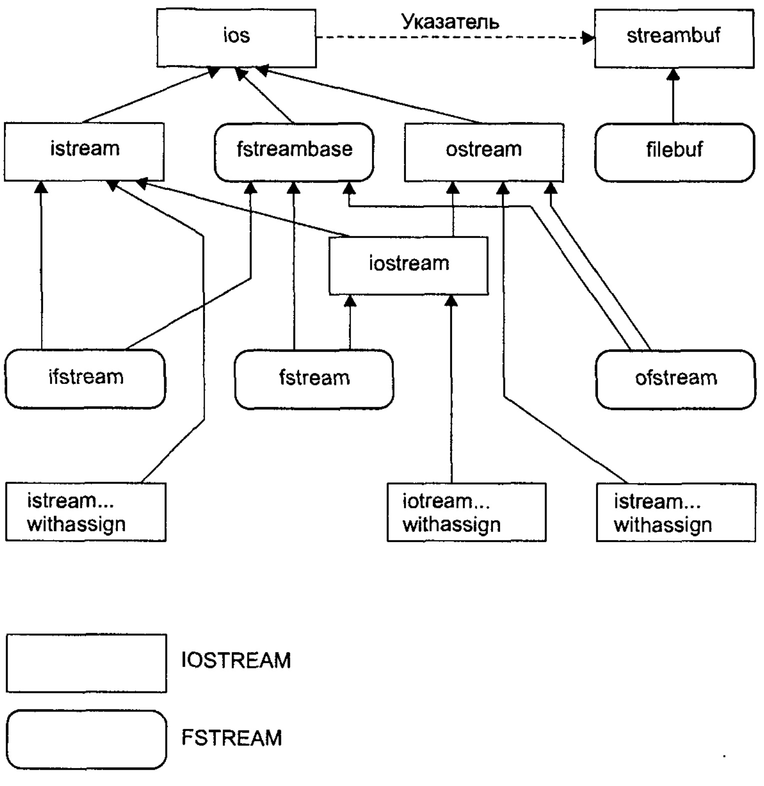


Рис. 12.1. Иерархия потоковых классов

На рис. 12.1 видно, что класс ios является базовым для всей иерархии. Он содержит множество констант и методов, общих для операций ввода/вывода любых видов. Некоторые из них, такие, как флаги форматирования showpoint и fixed, мы уже встречали. В классе ios есть также указатель на класс streambuf, значением которого является адрес текущего буфера памяти. С помощью буфера производится обмен данными (запись или чтение). Кроме того, ios содержит некоторые низкоуровневые программы для обработки этих данных. В обычной ситуации вам не нужно заботиться о классе streambuf, обращение к которому из других классов происходит автоматически.

Классы istream и ostream являются наследниками ios и предназначены для ввода и вывода соответственно. Класс istream содержит такие функции, как get(), getline(), read() и перегружаемую операцию извлечения (>>), тогда как класс ostream содержит функции put(), write() и перегружаемую операцию вставки (<<).

Класс iostream — наследник одновременно классов istream и ostream (пример множественного наследования). Его производные классы могут использоваться

при работе с такими объектами, как дисковые файлы, которые могут быть открыты одновременно для записи и чтения. Три класса — istream_withassign, ostream_withassign и iostream_withassign — являются наследниками istream, ostream и iostream соответственно. Они добавляют к этим классам операторы присваивания.

Приведенный ниже свод классов может показаться довольно абстрактным. Его можно пока пропустить, чтобы вернуться к нему при осуществлении каких-либо конкретных действий с использованием потоков.

Класс ios

Ios является дедушкой всех потоковых классов и обладает большинством особенностей, без которых работа с потоками была бы невозможна. Три его главных направления — это флаги форматирования, флаги ошибок и режим работы с файлами. Сначала мы обратимся к рассмотрению флагов форматирования, затем флагов состояния ошибки. Рассмотрение режима работы с файлами мы отложим до тех пор, пока не познакомимся с дисковыми файлами.

Флаги форматирования

Флаги форматирования — это набор определений enum (перечисляемого типа) в классе ios. Они работают переключателями, определяющими различные форматы и способы ввода/вывода. Мы не будем обсуждать подробно каждый флаг, так как одни мы уже использовали, а названия других говорят сами за себя. Некоторые мы обсудим позднее в этой же главе. В табл. 12.1 приводится полный список флагов форматирования.

Таблица 12.1. Флаги форматирования класса ios

Флаг	Значение
skipws	Пропуск пробелов при вводе
left	Выравнивание по левому краю
right	Выравнивание по правому краю
internal	Заполнение между знаком или основанием числа и самим числом
dec	Перевод в десятичную форму
oct	Перевод в восьмеричную форму
hex	Перевод в шестнадцатеричную форму
boolalpha	Перевод логического 0 и 1 соответственно в «false» и «true»
showbase	Выводить индикатор основания системы счисления (0 для восьмеричной, 0x для шестнадцатеричной)
showpoint	Показывать десятичную точку при выводе
uppercase	Переводить в верхний регистр буквы X, E и буквы шестнадцатеричной системы счисления (ABCDEF) (по умолчанию — в нижнем регистре)
showpos	Показывать «+» перед положительными целыми числами
scientific	Экспоненциальный вывод чисел с плавающей запятой
fixed	Фиксированный вывод чисел с плавающей запятой
unitbuf	Сброс потоков после вставки
stdio	сброс stdout, sterror после вставки

Есть несколько способов установки флагов форматирования, для каждого свои. Так как они являются компонентами класса ios, обычно к ним обращаются посредством написания имени класса и оператора явного задания (например, ios::skipws). Все без исключения флаги могут быть выставлены с помощью методов setf() и unsetf(). Взгляните на следующий пример:

```
cout.setf(ios::left);      //выравнивание текста по левому краю
cout >> "Этот текст выровнен по левому краю"
cout.unsetf(ios::left);    //вернуться к прежнему форматированию
```

Многие флаги могут быть установлены с помощью манипуляторов, давайте рассмотрим их.

Манипуляторы

Манипуляторы — это инструкции форматирования, которые вставляются прямо в поток. Мы уже видели некоторые из них, например endl, который посылает символ разделителя строк в поток и сбрасывает буфер:

```
cout << "Jedem Das Seine." << endl;
```

Мы встречали в программе SALEMON, глава 7 «Массивы и строки», манипулятор setiosflags(). Вспомним, как он выглядит:

```
cout  << setiosflags(ios::fixed)   //использовать
                                   //фиксированный вывод
<< setiosflags(ios::showpoint)     //показывать десятичную
                                   //точку
<< var;
```

Как показывают эти примеры, манипуляторы бывают двух видов — с аргументами и без. В табл. 12.2 сведены вместе все важные манипуляторы без аргументов.

Таблица 12.2. Манипуляторы ios без аргументов

Манипулятор	Назначение
ws	Включает пропуск пробелов при вводе
dec	Перевод в десятичную форму
oct	Перевод в восьмеричную форму
hex	Перевод в шестнадцатеричную форму
endl	Вставка разделителя строк и очистка выходного потока
ends	Вставка символа отсутствия информации для окончания выходной строки
flush	Очистка выходного потока
lock	Закрыть дескриптор файла
unlock	Открыть дескриптор файла

Вставляются эти манипуляторы прямо в поток. Например, чтобы вывести var в шестнадцатеричной форме, надо указать:

```
cout << hex << var;
```

Имейте в виду, что манипуляторы действуют только на те данные, которые следуют за ними в потоке, а не на те, которые находятся перед ними. На то он и поток. В табл. 12.3 представлены важнейшие из манипуляторов с аргументами. Для доступа к этим функциям вам потребуется заголовочный файл IOMANIP.

Таблица 12.3. Манипуляторы ios с аргументами

Манипулятор	Аргумент	Назначение
setw()	ширина поля (int)	Устанавливает ширину поля для вывода данных
setfill()	символ заполнения (int)	Устанавливает символ заполнения (по умолчанию, пробел)
setprecision()	точность (int)	Устанавливает точность (число выводимых знаков)
setiosflags()	Флаги форматирования (long)	Устанавливает указанные флаги форматирования
resetiosflags()	Флаги форматирования (long)	Сбрасывает указанные флаги форматирования

Функции

Класс ios содержит набор функций, с помощью которых можно выставлять флаги форматирования и выполнять некоторые другие действия. Таблица 12.4 содержит большинство этих функций, кроме тех, которые обрабатывают ошибки. К последним мы обратимся позднее.

Таблица 12.4. Функции ios

Функция	Назначение
ch=fill();	Возвращает символ заполнения (символ, которым заполняется неиспользуемая часть текстового поля; по умолчанию — пробел)
fill(ch);	Устанавливает символ заполнения
p=precision();	Возвращает значение точности (число выводимых знаков для формата с плавающей запятой)
precision(p);	Устанавливает точность p
w=width();	Возвращает текущее значение ширины поля (в символах)
width(w);	Устанавливает ширину текущего поля
setf(flags);	Устанавливает флаг форматирования (например, ios::left)
unsetf(flags);	Сбрасывает указанный флаг форматирования
setf(flags, field);	Очищает поле, затем устанавливает флаги форматирования

Эти функции вызываются для нужных потоковых объектов обычным способом — через точку. Например, чтобы установить ширину поля 12, можно написать:

```
cout.width(14);
```

Следующее выражение делает символом заполнения звездочку (как при тестировании печати):

```
cout.fill('*');
```

Можно использовать некоторые функции, чтобы манипулировать напрямую установкой флагов форматирования. Например, так можно установить выравнивание по левому краю:

```
cout.setf(ios::left);
```

Чтобы восстановить выравнивание по правому краю, напишем:

```
cout.unsetf(ios::left);
```

Версия setf() с двумя аргументами использует второй из них для сбрасывания всех флагов указанного типа (в *поле*). При этом установится флаг, указанный в качестве первого аргумента. Таким образом, проще сбросить флаги перед установкой нового. В табл. 12.5 показан способ обращения с этой функцией.

Например,

```
cout.setf(ios::left, ios::adjustfield);
```

сбрасывает все флаги, связанные с выравниванием текста, а затем устанавливает флаг left для выравнивания по левому краю.

Таблица 12.5. Версия setf() с двумя аргументами

Первый аргумент: устанавливаемые флаги	Второй аргумент: сбрасываемые флаги
dec, oct, hex	basefield
left, right, internal	adjustfield
scientific, fixed	floatfield

С использованием указанной техники можно создать способ форматированного ввода/вывода не только с клавиатуры и на дисплей, но, как мы увидим позднее в этой главе, также и для файлов.

Класс istream

Класс istream, наследник класса ios, выполняет специфические действия по вводу данных — извлечение. Очень легко спутать извлечение и связанное с ним действие по выводу данных — вставку. Рисунок 12.2 показывает разницу.

В табл. 12.6 собраны функции istream, которые вам могут пригодиться.

Таблица 12.6. Функции istream

Функция	Назначение
>>	Форматированное извлечение данных всех основных (и перегружаемых) типов из потока
get(ch)	Извлекает один символ в ch
get(str)	Извлекает символы в массив str до ограничителя '\n'
get(str, MAX)	Извлекает до MAX числа символов в массив

Функция	Назначение
get(str, DELIM)	Извлекает символы в массив str до указанного ограничителя (обычно '\n'). Оставляет ограничитель в потоке
get(str, MAX, DELIM)	Извлекает в массив str до MAX символов или до символа DELIM. Оставляет ограничитель в потоке
getline(str, MAX, DELIM)	Извлекает в массив str до MAX символов или символа DELIM. Извлекает ограничитель из потока
putback(ch)	Вставляет последний прочитанный символ обратно во входной поток
ignore(MAX, DELIM)	Извлекает и удаляет до MAX числа символов до ограничителя включительно (обычно '\n'). С извлеченными данными ничего не делает
peek(ch)	Читает один символ, оставляя его в потоке
count=gcount()	Возвращает число символов, прочитанных только что встретившимися вызовами get(), getline() или read()
read(str, MAX)	(Для файлов.) Извлекает вплоть до MAX числа символов в массив str до признака конца файла (EOF)
seekg()	Устанавливает расстояние (в байтах) от начала файла до файлового указателя
seekg(pos, seek_dir)	Устанавливает расстояние (в байтах) от указанной позиции в файле до указателя файла. seek_dir может принимать значения ios::beg, ios::cur, ios::end
pos=tellg(pos)	Возвращает позицию (в байтах) указателя файла от начала файла

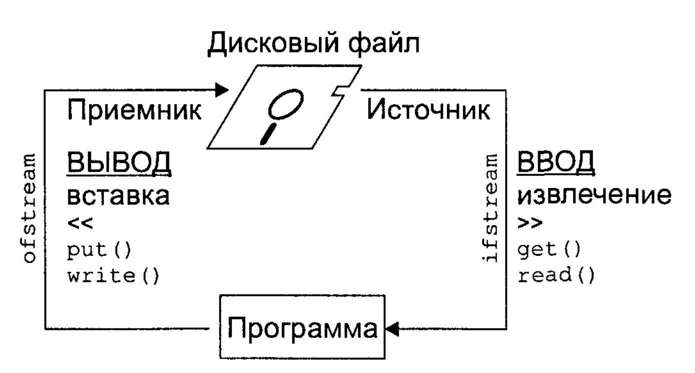


Рис. 12.2. Файловый ввод/вывод

Там уже встречались некоторые из этих функций, например get(). Большинство из них рассчитаны на работу с объектом cin, обычно представляющим собой поток данных, вводимых с клавиатуры. А последние четыре функции предназначены только для работы с дисковыми файлами.

Класс ostream

Класс ostream предназначен для вывода (вставки в поток) данных. Таблица 12.7 содержит наиболее общие методы этого класса. Как и в предыдущем случае, последние четыре функции работают только с файлами.

Таблица 12.7. Функции ostream

Функция	Назначение
<<	Форматированная вставка данных любых стандартных (и перегруженных) типов
put(ch)	Вставка символа ch в поток
flush()	Очистка буфера и вставка разделителя строк
write(str, SIZE)	Вставка SIZE символов из массива str в файл
seekp(position)	Устанавливает позицию в байтах файлового указателя относительно начала файла
seekp(position, seek_dir)	Устанавливает позицию в байтах файлового указателя относительно указанного места в файле. seek_dir может принимать значения ios::beg, ios::cur, ios::end
pos=tellp()	Возвращает позицию указателя файла в байтах

Классы iostream и _withassign

Класс iostream является порожденным по отношению к istream и ostream. Его единственное предназначение — выступать в качестве базового класса для других специфических классов вида iostream_withassign. У него нет собственных методов, кроме конструктора и деструктора. Его порожденные классы могут осуществлять как ввод, так и вывод данных.

Существуют три класса вида _withassign:

- istream_withassign — наследник istream;
- ostream_withassign — наследник ostream;
- iostream_withassign — наследник iostream.

Эти классы очень похожи на своих предков. Разница лишь в том, что в них, в отличие от породивших их классов, имеются перегружаемые операции присваивания, благодаря чему их объекты могут быть скопированы.

Для чего нужны потоковые классы с возможностью копирования и без таковой? Вообще-то, это не самая лучшая идея — копировать объекты потоковых классов. Причина заключается в том, что каждый такой объект ассоциирован с конкретным объектом streambuf, включающим в себя область памяти для хранения данных объекта. Если вместе с потоковым объектом копировать объект streambuf, возникнет некая неразбериха. Тем не менее бывают случаи, когда важно иметь возможность копировать потоки.

Соответственно, классы istream, ostream и iostream созданы так, что их объекты нельзя копировать — операторы присваивания и конструкторы копирования сделаны скрытыми. А их наследники с хвостиком _withassign имеют возможность копирования своих объектов.

Предопределенные потоковые объекты

В этой книге уже приводились примеры использования двух предопределенных потоковых объектов, порожденных классами вида _withassign: cin и cout. Обычно

они связаны с клавиатурой и монитором соответственно. Еще двумя предопределенными объектами являются cerr и clog:

- cin, объект istream_withassign, используется для операций ввода с клавиатуры;
- cout, объект ostream_withassign, используется для операций вывода на экран монитора;
- cerr, объект ostream_withassign, используется для сообщений об ошибках;
- clog, объект ostream_withassign, используется для ведения журнала.

Объект cerr часто используется для сообщений об ошибках и программной диагностики. Поток, посланный в него, немедленно выводится на экран, минуя буферизацию. Этим cerr отличается от cout. К тому же этот поток не может быть перенаправлен (подробнее об этом вы узнаете позже). Поэтому у вас есть небольшой шанс увидеть последнее сообщение неожиданно умершей программы. Другой объект, clog, похож на cerr в том, что также не может быть перенаправлен. Но его вывод проходит буферизацию.

Ошибки потоков

До сих пор в нашей книге мы использовали довольно прямолинейный подход к вводу/выводу, используя выражения вида

```
cout << "Доброе утро, страна!";
```

и

```
cin >> var;
```

Такой подход, как вы могли заметить, предполагает, что во время процесса ввода/вывода ничего нехорошего не случится. К сожалению, не всегда все проходит гладко, особенно в отношении ввода. Что будет, если наш дорогой пользователь введет строчку «девять» вместо числа 9, что будет, если он нажмет Enter, ничего не введя? А что случится, если произойдет скачок напряжения в электрической сети или просто возникнут какие-то неполадки с техникой? В процессе изучения этой темы мы коснемся всех этих проблем. Множество приемов, которые мы изучим, применимы и к файловому вводу/выводу.

Биты статуса ошибки

Флаги статуса ошибок потоков определяют компонент ios enum, который сообщает об ошибках, произошедших в операциях ввода/вывода. Все эти флаги собраны в табл. 12.8. Рисунок 12.3 показывает, как они выглядят. Для чтения (и даже установки) флагов могут использоваться различные функции ios, как показано в табл. 12.9.

Таблица 12.8. Флаги статуса ошибок

Название	Значение
goodbit	Ошибок нет (флаги не установлены, значение =0)
eofbit	Достигнут конец файла

продолжение ☞

Таблица 12.8 (*продолжение*)

Название	Значение
failbit	Операция не выполнена (пользовательская ошибка, преждевременный EOF)
badbit	Недопустимая операция (нет ассоциированного streambuf)
hardfail	Неисправимая ошибка

Таблица 12.9. Функции для флагов ошибок

Функция	Назначение
int=eof();	Возвращает true, если установлен флаг EOF
int=fail();	Возвращает true, если установлены флаги failbit, badbit или hardfail
int=bad();	Возвращает true, если установлены флаги badbit или hardfail
int=good();	Возвращает true, если ошибки не было
clear(int=0);	При использовании без аргумента снимает все флаги ошибок, в противном случае устанавливает указанный флаг, например clear(ios::failbit)

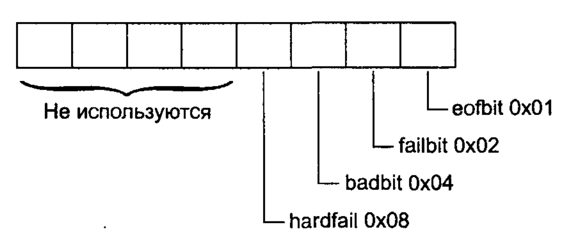


Рис. 12.3. Флаги состояния потока

Ввод чисел

Посмотрим, как обрабатываются ошибки при вводе чисел. Приводимый ниже подход применяется к числам, прочитанным с клавиатуры или с диска. Идея состоит в том, чтобы проверять значение goodbit, сообщать об ошибке, если его значение не равно true, и давать возможность пользователю ввести корректное число.

```
while(true)                  // Цикл до тех пор, пока
                             // ввод не будет корректным
  {
  cout <<"\nВведите целое число: ";
  cin >> i;
  if( cin.good() )           // если нет ошибок
    {
    cin.ignore(10, '\n');    // удалить разделитель строк
    break;
    }                        // выйти из цикла
  cin.clear();               // Очистить биты ошибок
  cout << "Неправильный ввод данных";
  cin.ignore(10, '\n');      // Удалить разделитель строк
  }
cout << "целое число: " << i; //целое без  ошибок
```

Самая общая ошибка, которая определяется по этой схеме, это ввод не цифр, а каких-либо символов (например, «девять» вместо «9»). Это приводит к уста-

новке флага failbit. Но определяются и системные ошибки, наиболее типичные при работе с дисковыми файлами.

Числа в формате с плавающей запятой (float, double и long double) могут анализироваться на предмет неправильного ввода так же, как и целые числа.

Переизбыток символов

Проблема превышения при вводе допустимого числа символов встречается довольно часто. Обычно это происходит при передаче входного потока с ошибками. Лишние символы все еще остаются в потоке после того, как ввод уже считается завершенным. Затем они передаются еще и в следующую операцию ввода, несмотря на то, что совершенно для этого не предназначены. Часто в конце остается символ новой строки, но иногда и другие. Чтобы избежать случайного извлечения из потока лишних символов, используется метод istream — функция ignore(MAX, DELIM). Она считывает и выкидывает вплоть до MAX числа символов, включая указанный ограничитель. В нашем примере выражение

```
cin.ignore(10. '\n');
```

приводит к считыванию до 10 символов, включая ограничитель '\n', и удалению их из входного потока.

Ввод при отсутствии данных

Символы, не несущие смысловой нагрузки, — пробелы и '\n' — обычно пропускаются при вводе данных. Это может привести к некоторым нежелательным побочным эффектам. Например, пользователи вместо ввода значения могут просто случайно нажать Enter, думая, что тем самым введут ноль, или просто запутавшись. Приведенный выше код также, как и выражение

```
cin >> i;
```

после нажатия клавиши Enter приведет к переводу курсора на новую строку, в то время как поток все еще будет ожидать ввода данных. В чем проблема с переводом курсора на новую строку? Дело в том, что, не увидев адекватной реакции на свои действия, пользователь может подумать, что компьютер вышел из строя. Кроме того, обычно повторные нажатия клавиши Enter приводят к тому, что курсор сдвигается все ниже и ниже, пока не дойдет до конца экрана, после чего экран начинает прокручиваться вверх. Хорошо еще, если пользователь и программа взаимодействуют в этаком телеграфном стиле, просто печатая друг другу сообщения. Но если нечто подобное произойдет в программе, в которой используется графика, основанная на тексте (как в программе ELEV из главы 13 «Многофайловые программы»), прокручивание окна приведет к полной неразберихе на экране.

Таким образом, важно иметь возможность дать команду входному потоку *не* игнорировать разделители. Это делается с помощью флага skipws:

```
cout << "\nВведите целое число: ";
cin.unsetf(ios::skipws);//не игнорировать разделители
```

```
cin >> i;
if( cin.good() )
  {
  //ошибок нет
  }
//ОШИБКА!
```

Уж теперь если пользователь и нажмет Enter, забыв ввести данные, то будет установлен флаг failbit и тем самым сгенерирован признак ошибки. После этого можно попросить пользователя ввести данные повторно или перемещать курсор так, чтобы экран не прокручивался.

Ввод строк и символов

Пользователь не может, на самом деле, сделать больших ошибок при вводе строк и символов, так как ввод любых данных, даже чисел, можно интерпретировать, как ввод строк. Тем не менее, если поток пришел из дискового файла, символы и строки должны проверяться на наличие ошибок, так как не всегда все бывает ладно с признаком окончания файла (EOF). В отличие от ситуации ввода чисел здесь вам как раз необходимо игнорировать разделители.

Отладка примера с английскими расстояниями

Давайте взглянем на программу, в которой ввод пользователем данных в класс Distance проверяется на наличие ошибок. Забудем на время про перевод расстояний из одних единиц в другие и про работу с ними. Пусть наша программа будет просто получать от пользователя данные в футах и дюймах и выводить их на экран. Если же пользователь ошибается, нужно отклонить введенные данные, объяснить ему, где он ошибся, и попросить ввести данные заново.

Программа очень проста, за исключением того, что метод getdist() расширен и направлен на поддержку обработки ошибок. Частично эта программа использует подход, показанный в приведенном выше коде. Кроме того, мы еще ввели выражение, которое следит за тем, чтобы пользователь не вводил футы в формате чисел с плавающей запятой. Это действительно важно, так как дюймы могут иметь дробную часть, а футы — нет, и пользователь может слегка запутаться.

Обычно, ожидая ввод целочисленных значений, оператор извлечения просто прерывает свою работу, увидев точку в десятичной дроби. При этом сообщение об ошибке не выдается. Мы же хотим узнать о совершенной ошибке, поэтому будем считывать значение числа футов не в виде int, а в виде строки. Затем мы проверяем введенную строку при помощи созданной нами функции isFeet(), возвращающей true, если число футов введено корректно. Для одобрения нашей функцией введенное число должно содержать только цифры, и его значение должно лежать в пределах от –999 до 999 (мы просто предполагаем, что более далекие расстояния измерять не будем). Если с введенными футами все в порядке, конвертируем строку в int с помощью стандартной библиотечной функции atoi(). Число дюймов вполне может быть дробным. Мы будем проверять только их значения — они должны лежать в пределах от 0 до 12.0. Кроме того,

будем отслеживать наличие флагов ошибок ios. В общем случае будет установлен failbit при вводе каких-либо символов вместо числа. Приведем листинг ENGLERR[1].

Листинг 12.1. Программа ENGLERR

```
// englerr.cpp
// контроль ввода данных для класса английских расстояний
#include <iostream>
#include <string>
#include <cstdlib>       // для atoi(), atof()
using namespace std;
int isFeet(string);      //объявление
///////////////////////////////////////////////////////////
class Distance           // Класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
    Distance()                 // конструктор (без аргументов)
      { feet = 0; inches = 0.0; }
    Distance(int ft, float in) // конструктор (2 арг.)
      { feet = ft; inches = in; }
    void showdist()            // вывод расстояния
      { cout << feet << "\'-" << inches << '\"';}
    void getdist();            // запросить длину
                               // у пользователя
  };
//---------------------------------------------------------
void Distance::getdist()    // получение длины от пользователя
  {
  string instr;             // для входной строки

  while(true)               // цикл, пока футы
                            // не будут правильными
    {
    cout << "\n\nВведите футы: ";
    cin.unsetf(ios::skipws);// не пропускать
                            // разделители
    cin >> instr;           // получить футы как строку
    if( isFeet(instr) )     // правильное значение?
      {      //да
      cin.ignore(10, '\n'); // съесть символы,
                            // включая разделитель строк
      feet = atoi( instr.c_str() ); // перевести
                                    // значение в целочисленное
      break;                // выход из цикла 'while'
      }                     // нет, не целое
    cin.ignore(10, '\n');   // съесть символы, включая
                            // разделитель строк
    cout << "Футы должны быть целыми < 1000\n";
    } //конец цикла while для футов

  while(true)               // цикл проверки дюймов
    {
    cout << "Введите дюймы: ";
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
    cin.unsetf(ios::skipws);// не пропускать
                            // разделители
    cin >> inches;          // получить дюймы (тип float)
    if(inches>=12.0 || inches<0.0)
      {
      cout << "Дюймы должны быть между 0.0 и 11.99\n";
      cin.clear(ios::failbit); //"искусственно"
                                //установить флаг ошибки
      }
    if( cin.good() )        // все ли хорошо с cin
      { //(обычно вводят не цифры)
      cin.ignore(10, '\n'); // съесть разделитель
      break;                // Ввод корректный, выйти из 'while'
      }
    cin.clear();            // ошибка; очистить статус ошибки
    cin.ignore(10, '\n'); // съесть символы с разделителем
    cout << "Неверно введены дюймы\n";  //заново
    }  //конец while для дюймов
  }
//-----------------------------------------------------------
int isFeet(string str)      // true если введена строка
  {                         // с правильным значением футов
  int slen = str.size();    // получить длину
  if(slen==0 || slen > 5)   // не было или слишком много
                            // данных
    return 0;               // не целое
  for(int j=0; j<slen; j++) // проверить каждый символ
                            // если не цифра или минус
  if((str[j] < '0' || str[j] > '9')&&str[j] != '-')
      return 0;             // строка неправильных футов
  double n = atof( str.c_str() ); // перевод в double
  if( n<-999.0 || n>999.0 )  // вне допустимых значений?
    return 0;                // если да, неправильные футы
  return 1;                  // правильные футы
  }
/////////////////////////////////////////////////////////////
int main()
  {
  Distance d;               //создать объект Distance
  char ans;
  do
    {
    d.getdist();            // получить его значение
    cout << "\nРасстояние = ";
    d.showdist();           // вывести его
    cout << "\nЕще раз (y/n)? ";
    cin >> ans;
    cin.ignore(10,'\n' ); // съесть символы и
                          // разделитель
    } while(ans != 'n');  // цикл до 'n'
  return 0;
  }
```

Мы использовали еще одну хитрость: установили флаг ошибки вручную. Тем самым мы проверили, входят ли введенные дюймы в диапазон допустимых значений. Если нет, мы устанавливает failbit с помощью

```
cin.clear(ios::failbit);//установить failbit
```

При проверке на наличие ошибок с помощью функции cin.good() флаг failbit будет обнаружен и выведется сообщение об ошибке.

Потоковый ввод/вывод дисковых файлов

Большинству программ требуется сохранять данные на диске и считывать их. Работа с дисковыми файлами подразумевает наличие еще одного набора специальных классов: ifstream для ввода и ofstream для вывода. Класс fstream осуществляет и ввод, и вывод. Объекты этих классов могут быть ассоциированы с дисковыми файлами, а их методы — использоваться для обмена данными с ними.

Вернемся к рис. 12.1. На нем видно, что класс ifstream является наследником класса istream, ofstream — класса ostream, fstream — класса iostream. Эти родительские классы, в свою очередь, являются наследниками класса ios. Такая иерархия вполне логична — классы, ориентированные на работу с файлами, могут использовать методы более общих классов. К тому же файловые классы используют принцип множественного наследования, будучи наследниками еще и класса fstreambase. Этот класс содержит объект класса filebuf, являющегося файловым буфером, а также ассоциированные методы, унаследованные от более общего класса streambuf. Обычно программисту не приходится заботиться об этих буферных классах.

Классы ifstream, ofstream и fstream объявлены в файле FSTREAM.

Программисты на С наверняка обратят внимание на то, что подход к дисковому вводу/выводу в С++ оказывается совсем иным. Старые функции языка С, такие, как fread() и fwrite()в С++, конечно, работают, но они не так хорошо вписываются в концепцию объектно-ориентированной среды программирования. Новый подход, предлагаемый С++, гораздо прозрачнее и проще в использовании. Кстати говоря, остерегайтесь случайного перемешивания старых функций С с потоками С++. Они не всегда в хороших отношениях друг с другом, хотя есть возможность заставить их жить дружно.

Форматированный файловый ввод/вывод

При форматированном вводе/выводе числа хранятся на диске в виде серии символов. Таким образом, число 6,02 вместо того, чтобы храниться в виде четырехбайтного значения типа float или восьмибайтного double, хранится в виде последовательности символов '6', ',', '0', '2'. Это странно с точки зрения экономии памяти, особенно в случае многоразрядных чисел, но зато легко применяется на практике и в некоторых ситуациях гораздо удобнее. Слава Богу, что хоть символы и строки хранятся в файлах в более или менее привычной форме.

Запись данных

Следующая программа демонстрирует запись символа, целого числа, числа типа double и двух объектов типа string в дисковый файл. Вывод на экран не производится. Приведем листинг программы FORMATO.

Листинг 12.2. Программа FORMATO

```
// formato.cpp
// Форматированный вывод в файл с использованием <<
#include <fstream>        // для файлового ввода/вывода
#include <iostream>
#include <string>
using namespace std;

int main()
  {
  char ch = 'x';
  int j = 77;
  double d = 6.02;
  string str1 = "Kafka";  // строки без
  string str2 = "Proust"; // пробелов

  ofstream outfile("fdata.txt"); //создать объект ofstream

  outfile << ch           // вставить (записать) данные
       << j
       << ' '             // пробелы между числами
       << d
       << str1
       << ' '             // пробелы между строками
       << str2;
  cout << "Файл записан\n";
  return 0;
  }
```

Здесь мы определили объект outfile в качестве компонента класса ofstream. В то же время мы инициализировали его файлом FDATA.TXT. Инициализация резервирует для дискового файла с данным именем различные ресурсы и получает доступ (или *открывает файл*) к нему. Если файл не существует, он создается. Если файл уже существует, он переписывается — новые данные в нем заменяют старые. Объект outfile ведет себя подобно cout из предыдущих программ, поэтому можно использовать операцию вставки (<<) для вывода переменных любого из стандартных типов в файл. Это так замечательно работает потому, что оператор вставки перегружен в классе ostream, который является родительским для ofstream.

Когда программа завершается, объект outfile вызывает свой деструктор, который закрывает файл, так что нам не приходится делать это явным образом.

Есть несколько потенциальных проблем с форматированным выводом в дисковые файлы. Во-первых, надо разделять числа (77 и 6,02, например) нечисловыми символами. Поскольку они хранятся в виде последовательности символов, а не в виде полей фиксированной длины, это единственный шанс узнать при извлечении, где кончается одно и начинается другое число. Во-вторых, между строками должны быть разделители — по тем же причинам. Это подразумевает, что внутри строки не может быть пробелов. В этом примере для разделения данных мы использовали пробел в обоих случаях. Радует то, что символы не нуждаются в разделителях, хотя они и являются данными фиксированной длины.

Посмотреть на результаты работы программы FORMATO можно, открыв файл FDATA.TXT с помощью программы WORDPAD или команды DOS TYPE.

Чтение данных

Прочитать файл, созданный программой FORMATO, можно с использованием объекта ifstream, инициализированного именем файла. Файл автоматически открывается при создании объекта. Затем данные из него можно считать с помощью оператора извлечения (>>).

Приведем листинг программы FORMATI, которая считывает данные из файла FDATA.TXT.

Листинг 12.3. Программа FORMATI

```
// formati.cpp
// форматированное чтение из файла с помощью >>
#include <fstream>              // для файлового ввода/вывода
#include <iostream>
#include <string>
using namespace std;

int main()
  {
  char ch;
  int j;
  double d;
  string strl;
  string str2;

  ifstream infile("fdata.txt"); // создать объект ifstream
                                // извлечь (прочесть) из него данные
  infile >> ch >> j >> d >> strl >> str2;

  cout << ch << endl            // вывести данные
      << j << endl
      << d << endl
      << strl << endl
      << str2 << endl;
  return 0;
  }
```

Объект ifstream, который мы назвали infile, действует примерно так же, как cin в предыдущих программах. Если мы корректно вставляли данные в файл, то извлечь их не составит никаких проблем. Мы просто сохраняем их в соответствующих переменных и выводим на экран. Результаты работы программы:

```
x
77
6.02
Kafka
Proust
```

Разумеется, числа приводятся обратно к своему двоичному представлению, чтобы с ними можно было работать в программе. Так, 77 сохраняется в переменной j типа int, это уже теперь не две семерки символьного типа. 6,02 сохраняется в переменной типа double.

Строки с пробелами

Подход, продемонстрированный в последнем примере, не позволяет обрабатывать строки с char*, содержащие пробелы. Для того чтобы эта ситуация изменилась, после каждой строки нужно писать специальный символ-ограничитель и использовать функцию getline() вместо оператора извлечения. Наша следующая маленькая программа OLINE выводит строки с пробелами. Посмотрите, как это сделано.

Листинг 12.4. Программа OLINE

```
// oline.cpp
// файловый вывод строк с пробелами
#include <fstream>              // для операций
                                // файлового ввода/вывода
using namespace std;

int main()
  {
  ofstream outfile("TEST.TXT"); // создать выходной файл
                                // отправить текст в файл
  outfile << "Приходит март. Я сызнова служу.\n";
  outfile << "В несчастливом кружении событий \n";
  outfile << "изменчивую прелесть нахожу \n";
  outfile << "в смешеньи незначительных наитий.\n";
  return 0;
  }
```

Когда вы запустите программу, строки стихотворения И. Бродского будут записаны в файл. Каждая строка заканчивается символом разделения строк ('\n'). Обратите внимание, это строки типа char*, а не объекты класса string. Многие потоковые операции гораздо легче производить с этим типом данных.

Чтобы извлечь строки из файла, мы создадим ifstream и станем читать строчку за строчкой функцией getline(), которая является методом класса istream. Эта функция считывает все символы, включая разделители, пока не дойдет до специального символа '\n', затем помещает результат чтения в буфер, переданный ей в качестве аргумента. Максимальный размер буфера передается с помощью второго аргумента. Его содержимое выводится на экран после считывания каждой строки.

Листинг 12.5. Программа ILINE

```
// iline.cpp
// Файловый ввод (извлечение из файла) строк
#include <fstream>              // для файловых функций
#include <iostream>
using namespace std;

int main()
  {
  const int MAX = 80;           // размер буфера
  char buffer[MAX];             // буфер символов
  ifstream infile("TEST.TXT");  // создать входной файл
  while( !infile.eof() )        // цикл до EOF
    {
```

```
    infile.getline(buffer, MAX); // читает строку текста
    cout << buffer << endl;      // и выводит ее
    }
  return 0;
}
```

В результате работы программы на экране появятся все та же строфа из стихотворения, которую мы записали в файл TEST.TXT в процессе работы программы OLINE. Программе не дано знать, сколько всего строчек в тексте, поэтому она продолжает чтение до тех пор, пока не встретит признак окончания файла (EOF). Учтите, что эту программу нельзя применять для чтения произвольных файлов, — каждая строка текста должна заканчиваться символом '\n'. При попытке прочитать файл иной структуры программа зависнет.

Определение признака конца файла (EOF)

Итак, объекты порожденных из ios классов содержат флаги статуса ошибок, с помощью которых можно проверить результат выполнения операций. При чтении файла небольшими порциями, как в этом примере, мы рано или поздно наткнемся на условие окончания файла. Сигнал EOF посылается в программу операционной системой, когда больше нет данных для чтения. В программе ILINE это условие встречалось в выражении

```
while( !infile.eof() )    //Пока не EOF
```

Это все замечательно, но надо учитывать, что, проверяя конкретный флаг признака окончания файла, мы не проверяем все остальные флаги ошибок. А failbit и badbit тоже могут возникнуть при работе программы, хотя это случается довольно редко. Чтобы проверять все, что можно, мы изменим условие цикла:

```
while( infile.good() )    //Пока нет ошибок
```

Можно также проверять поток напрямую. Любой потоковый объект, например infile, имеет значение, которое может тестироваться на предмет выявления наиболее распространенных ошибок, включая EOF. Если какое-либо из условий ошибки имеет значение true, объект возвращает ноль. Если все идет хорошо, объект возвращает ненулевое значение. Это значение на самом деле является указателем, но возвращаемый «адрес» никакой смысловой нагрузки не несет — он просто должен быть нулем либо не нулем. Поэтому запишем еще один вариант цикла while:

```
while( infile )           //Пока нет ошибок
```

На вид это выражение, конечно, очень простое, но непосвященным, скорее всего, будет непонятно, что оно делает.

Ввод/вывод символов

Функции put() и get(), являющиеся методами ostream и istream соответственно, могут быть использованы для ввода и вывода единичных символов. В программе OCHAR строка выводится именно посимвольно.

Листинг 12.6. Программа OCHAR

```
// ochar.cpp
// Посимвольный файловый вывод
#include <fstream>              // для файловых функций
#include <iostream>
#include <string>
using namespace std;

int main()
  {
  string str = "Время - великий учитель, но, увы, "
          "оно убивает своих учеников.  Берлиоз";

  ofstream outfile("TEST.TXT");   // Создать выходной файл
  for(int j=0; j<str.size(); j++) // каждый символ
    outfile.put( str[j] );        // записывать в файл
  cout << "Файл записан\n";
  return 0;
  }
```

В этой программе объект ofstream создается так же, как в программе OLINE. Длина объекта класса string по имени str находится с помощью метода size(), а символы выводятся в цикле for функцией put(). Афоризм Гектора Берлиоза (композитора XIX века) записывается в файл TEST.TXT. Считываем и выводим этот файл с помощью программы ICHAR.

Листинг 12.7. Программа ICHAR

```
// ichar.cpp
// Посимвольный файловый ввод
#include <fstream>              // для файловых функций
#include <iostream>
using namespace std;

int main()
  {
  char ch;                      // символ для считывания
  ifstream infile("TEST.TXT");  // входной файл
  while( infile )               // читать до EOF или ошибки
    {
    infile.get(ch);             // считать символ
    cout << ch;                 // и вывести его
    }
  cout << endl;
  return 0;
  }
```

В этой программе используется функция get(). Чтение производится до признака окончания файла (или возникновения ошибки). Каждый прочитанный символ выводится с помощью cout, поэтому на экран в результате работы программы будет выведен весь афоризм.

Есть и другой способ читать символы из файла — использовать функцию rdbuf() (она является методом класса ios). Функция возвращает указатель на объект класса streambuf (или filebuf), ассоциированный с потоковым объектом. В этом объекте находится буфер символов, считанных из потока, поэтому можно использовать указатель на него в качестве объекта данных.

Листинг 12.8. Программа ICHAR2

```
// ichar2.cpp
// Файловый ввод символов
#include <fstream>            // для файловых функций
#include <iostream>
using namespace std;

int main()
  {
  ifstream infile("TEST.TXT"); // создать входной файл

  cout << infile.rdbuf();      // передать его буфер в cout
  cout << endl;
  return 0;
  }
```

Результат работы этой программы совпадает с ICHAR. Можно взять с полки пирожок за написание самой короткой программы чтения из файла. Обратите внимание на то, что функция rdbuf() сама знает, что следует прекратить работу при достижении EOF.

Двоичный ввод/вывод

Форматированный файловый ввод/вывод чисел целесообразно использовать только при их небольшой величине и малом количестве. В противном случае, конечно, гораздо эффективнее использовать двоичный ввод/вывод, при котором числа хранятся таким же образом, как в ОП компьютера, а не в виде символьных строк. Целочисленные значения занимают 4 байта, тогда как текстовая версия числа, например «12345», занимает 5 байтов. Значения типа float также всегда занимают 4 байта. А форматированная версия «6,02314e13» занимает 10 байтов.

В следующем примере показано, как в бинарном виде массив целых чисел записывается в файл и читается из него. При этом используются две функции — write() (метод класса ofstream), а также read() (метод ifstream). Эти функции думают о данных в терминах байтов (тип char). Им все равно, как организованы данные, что они собой представляют, — они просто переносят байты из буфера в файл и обратно. Параметрами этих функций являются адрес буфера и его длина. Адрес должен быть вычислен с использованием reinterpret_cast относительно типа char*. Вторым параметром является длина в байтах (а не число элементов данных в буфере).

Листинг 12.9. Программа BINIO

```
// binio.cpp
// Двоичный ввод/вывод целочисленных данных
#include <fstream>          // для файловых потоков
#include <iostream>
using namespace std;
const int MAX = 100;        // размер буфера
int buff[MAX];              // буфер для целых чисел

int main()
  {
  for(int j=0; j<MAX; j++)  // заполнить буфер данными
    buff[j] = j;            // (0, 1, 2, ...)
```

продолжение ⇨

Листинг 12.9 (*продолжение*)

```
                           // создать выходной поток
ofstream os("edata.dat", ios::binary);
                           // записать в него
os.write(reinterpret_cast<char*>(buff), MAX*sizeof(int) );
os.close();                // должен закрыть его

for(j=0; j<MAX; j++)       // стереть буфер
  buff[j] = 0;
                           // создать входной поток
ifstream is("edata.dat", ios::binary);
                           // читать из него
is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );

for(j=0; j<MAX; j++)       // проверка данных
  if( buff[j] != j )
    { cerr << "Некорректные данные!\n"; return 1; }
cout << "Данные корректны\n";
return 0;
}
```

При работе с бинарными данными в качестве второго параметра write() и read() следует использовать ios::binary. Это необходимо по той причине, что текстовый режим, используемый по умолчанию, допускает несколько вольное обращение с данными. Например, специальный символ '\n' занимает два байта (на самом деле это и есть два действия — перевод каретки и перевод строки). Это делает текст более удобным для чтения в DOS утилитами типа TYPE, но для бинарных данных такой подход не годится вовсе, так как любой байт, которому не повезло иметь ASCII-код 10, переводится двумя байтами. Аргумент ios::binary — типичный пример *бита состояния*. Мы еще будем говорить об этом при обсуждении функции open() немного позднее в этой главе.

Оператор reinterpret_cast

Программа BINIO (как и многие последующие) использует оператор reinterpret_cast для того, чтобы буфер данных типа int выглядел для функций read() и write() как буфер типа char.

```
is.read(reinterpret_cast<char*>(buff), MAX*sizeof(int) );
```

reinterpret_cast как бы говорит компилятору: «Я знаю, что тебе это не понравится, и все-таки я это сделаю». Он изменяет тип данных в определенной области памяти, совершенно не задумываясь о том, имеет это смысл или нет. Поэтому вопрос целесообразности использования этого оператора остается целиком на совести программиста.

Можно использовать reinterpret_cast для превращения указателей в данные типа int и обратно. Это небезопасное занятие, но порой оно необходимо.

Закрытие файлов

До сих пор в наших примерах не нужно было вручную закрывать файлы — это делалось автоматически при окончании работы с ними. Соответствующие объек-

ты запускали деструкторы и закрывали ассоциированные файлы. Но в программе BINIO оба потока, входной is и выходной os, связаны с одним и тем же файлом EDATA.DAT, поэтому первый поток должен быть закрыт до того, как откроется второй. Для этого используется функция close().

Можно запускать эту функцию каждый раз при закрытии файла, не полагаясь на деструктор потока.

Объектный ввод/вывод

Так как C++ — это все-таки объектно-ориентированный язык, было бы интересно узнать, как происходит запись объектов в дисковые файлы и чтение из них. Следующий пример демонстрирует этот процесс. Класс person, использовавшийся ранее в нескольких наших программах (например, в VIRTPERS, глава 11, раздел «Виртуальные функции»), создает объекты, с которыми мы сейчас будем работать.

Запись объекта на диск

При записи объекта мы обычно используем бинарный режим. При этом на диск записывается та же битовая конфигурация, что хранится в памяти. Это придает уверенности в том, что данные объекта будут обработаны корректно. В приводимой ниже программе OPERS у пользователя запрашивается информация об объекте класса person, который потом записывается в файл PERSON.DAT.

Листинг 12.10. Программа OPERS

```
// opers.cpp
// Сохранение обьекта в файле
#include <fstream>        // для файловых потоков
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person              // класс person
  {
  protected:
    char name[80];        // имя человека
    short age;            // возраст
  public:
    void getData()        // получить данные о человеке
      {
      cout << "Введите имя: "; cin >> name;
      cout << "Введите возраст: "; cin >> age;
      }
  };
////////////////////////////////////////////////////////////////
int main()
  {
  person pers;            // создать обьект
  pers.getData();         // получить данные
                          // создать обьект ofstream
  ofstream outfile("PERSON.DAT", ios::binary);
                          // записать в него
  outfile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
  return 0;
  }
```

Метод getData() класса person вызывается для того, чтобы запросить у пользователя информацию, которая помещается в объект pers. Вот простой пример взаимодействия:

```
Введите имя: Артур Хайн
Введите возраст: 60
```

Содержимое данного объекта записывается на диск с помощью функции write(). Для нахождения длины данных объекта pers используется оператор sizeof.

Чтение объекта с диска

Для чтения пригодится метод read(), что продемонстрировано в следующем листинге.

Листинг 12.11. Программа IPERS

```
// ipers.cpp
// Чтение обьекта из файла
#include <fstream>       // для файловых потоков
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
class person             // класс person
  {
  protected:
    char name[80];       // Имя человека
    short age;           // его возраст
  public:
    void showData()      // вывести данные
      {
      cout << "Имя: " << name << endl;
      cout << "Возраст: " << age << endl;
      }
  };
//////////////////////////////////////////////////////////
int main()
  {
  person pers;           // переменная типа person
  ifstream infile("PERSON.DAT", ios::binary);// создать поток
                         // чтение потока
  infile.read( reinterpret_cast<char*>(&pers), sizeof(pers));
  pers.showData();       // вывести данные
  return 0;
  }
```

В результате работы программы будет выведено на экран все то, что было помещено в файл PERSON.DAT программой OPERS:

```
Имя: Артур Хайн
Возраст: 60
```

Совместимость структур данных

Для корректной работы программы чтения и записи объектов (такие, как OPERS и IPERS) должны иметь в виду один класс объектов. Например, объекты класса person имеют длину ровно 82 байта, из которых 80 отведено под имя человека,

2 — под возраст в формате short. Если бы программы не знали длину полей, то одна из них не смогла бы корректно прочитать из файла то, что записала другая.

Несмотря на то что классы person в OPERS и IPERS имеют одинаковые компонентные данные, у них могут быть совершенно разные методы. Скажем, в первой программе есть функция getData(), тогда как во второй — showData(). Не имеет значения, какие используются методы в классах, — они в файл вместе с данными не записываются. Это для данных важен единый формат, а разногласие между методами ни к каким последствиям не приводит. Впрочем, это утверждение справедливо только для обычных классов, в которых не используются виртуальные функции.

Если вы пишете в файл и читаете объекты производных классов, необходимо соблюдать большую осторожность. В этих объектах есть загадочное число, которое ставится перед началом их области данных в памяти. Оно помогает идентифицировать класс объекта при использовании виртуальных функций. Когда вы записываете объект в файл, это число записывается наряду с другими данными. Если меняются методы класса, идентификатор также изменяется. Если в классе имеются виртуальные функции, то при чтении объекта с теми же данными, но другими методами возникнут большие трудности. Отсюда мораль: класс, использующийся для чтения объекта, должен быть *идентичен* классу, использовавшемуся при его записи.

Можно даже не пытаться произвести дисковый ввод/вывод объектов, компонентами которых являются указатели. Легко догадаться, что значения указателей не будут корректными при чтении объекта в другую область памяти.

Ввод/вывод множества объектов

Предыдущие программы записывали и читали только один объект за раз. В следующем примере в файл записывается произвольное число объектов. Затем они считываются, и на экран выводится целиком содержимое файла.

Листинг 12.12. Программа DISKFUN

```
// diskfun.cpp
// Чтение из файла и запись нескольких обьектов
#include <fstream>      // для файловых потоков
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class person            // класс person
  {
  protected:
    char name[80];      // имя человека
    int age;            // его возраст
  public:
    void getData()      // получить данные о человеке
      {
      cout << "\n  Введите имя: "; cin >> name;
      cout << "  Введите возраст: "; cin >> age;
      }
    void showData()     // вывод данных о человеке
```

продолжение ⇒

Листинг 12.12 (*продолжение*)

```
      {
      cout << "\n  Имя: " << name;
      cout << "\n  Возраст: " << age;
      }
  };
////////////////////////////////////////////////////////////
int main()
  {
  char ch;
  person pers;          // создать объект person
  fstream file;         // создать входной/выходной файл
                        // открыть для дозаписи
  file.open("GROUP.DAT", ios::app | ios::out |
                       ios::in | ios::binary );
  do                    // данные от пользователя - в файл
    {
    cout << "\nВведите данные о человеке:";
    pers.getData();     // получить данные
                        // записать их в файл
    file.write( reinterpret_cast<char*>(&pers), sizeof(pers) );
    cout << "Продолжить ввод (y/n)? ";
    cin >> ch;
    }
  while(ch=='y');       // выход по 'n'
  file.seekg(0);        // поставить указатель на начало файла
                        // считать данные о первом человеке
  file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
  while( !file.eof() )  // Выход по EOF
    {
    cout << "\nПерсона:"; //вывести данные
    pers.showData();      //считать данные о следующем
    file.read(reinterpret_cast<char*>(&pers),sizeof(pers));
    }
  cout << endl;
  return 0;
  }
```

Покажем пример взаимодействия при работе программы DISKFUN. Предполагается, что она уже запускалась и были введены и записаны в файл данные о двух персонах.

```
Введите данные о человеке:
Введите имя:Екатерина Железнякова
Введите возраст:17
Продолжить ввод (y/n)? n
Персона:
Имя:Татьяна Васильева
Возраст:17
Персона:
Имя:Артур Хайн
Возраст:60
Персона:
Имя:Екатерина Железнякова
Возраст:17
```

В этом примере к файлу добавляется один объект, затем все содержимое, состоящее из трех объектов, выводится на экран.

Класс fstream

До сих пор мы создавали файловые объекты, предназначенные либо для ввода, либо для вывода. В программе DISKFUN сделана попытка создать файл, который может быть использован одновременно как входной и выходной. Это должен быть объект класса fstream, наследник класса iostream, порожденного классами istream и ostream, чтобы была возможность поддержки одновременно и ввода, и вывода.

Функция open()

Раньше мы создавали файловый объект и инициализировали его таким выражением:

```
ofstream outfile("TEST.TXT");
```

В программе DISKFUN использован следующий подход: одним выражением мы создаем файл, а другим его открываем, используя функцию open() (метод класса fstream). Так полезно делать, когда есть вероятность ошибки при открытии. Можно один раз создать потоковый объект, а затем циклически пытаться открыть его, не ломая голову над созданием каждый раз нового.

Биты режимов

Мы уже встречали бит режима ios::binary. При написании функции open() использовали еще несколько. Теперь сведем их все вместе в табл. 12.10. Биты режимов, определенные в ios, определяют различные методы открытия потоковых объектов.

Таблица 12.10. Биты режимов

Бит режима	Результат
in	Открытие для чтения (по умолчанию для ifstream)
out	Открытие для записи (по умолчанию для ofstream)
ate	Чтение, начиная с конца файла (AT End)
app	Запись, начиная с конца файла (AAPend)
trunc	Обрезать файл до нулевой длины, если он уже существует (TRUNCate)
nocreate	Не открывать несуществующий файл
noreplace	Не открывать для вывода существующий файл, если не установлены ate или app
binary	Открыть в бинарном (не текстовом) режиме

В DISKFUN использовался бит ios::app, потому что нам требовалось сохранить все, что было записано в файл до этого. То есть можно записать что-нибудь в файл, завершить программу, запустить ее заново и продолжать записывать данные, сохранив при этом результаты предыдущей сессии. ios::in и ios::out мы используем потому, что хотим осуществлять одновременно и ввод, и вывод. ios::binary также необходим, потому что мы записываем бинарные объекты. Вертикальные палочки между флагами нужны для того, чтобы из битов сформировалось единое целое число. При этом несколько флагов будут применяться одновременно.

За один раз в файл записывается один объект person с помощью функции write(). После окончания записи мы хотим прочитать файл целиком. Для этого нужно вначале установить указатель файла на начало. Этим занимается функция seekg(), к которой мы обратимся в следующем параграфе. После этого мы уже уверены, что чтение начнется с начала файла. Затем в цикле while мы считываем объект из файла и выводим его на экран.

Это продолжается до тех пор, пока не будут прочитаны все объекты класса person, — состояние, определяемое флагом ios::eofbit.

Указатели файлов

У каждого файлового объекта есть два ассоциированных с ним значения, называемые *указатель чтения* и *указатель записи*. Их также называют *текущая позиция чтения* и *текущая позиция записи*. Или, если так лучше для восприятия, просто *текущая позиция*. Эти значения определяют номер байта относительно начала файла, с которого будет производиться чтение или запись. (Слово «указатель» в этом контексте не следует путать с обычными указателями C++, используемыми в качестве адресных переменных.)

Часто требуется начинать чтение файла с начала и продолжать до конца. Во время записи бывает нужно начинать с начала, удаляя существующую информацию, или с конца, для чего файл следует открывать с флагом ios::app. Это действия, которые выполняются по умолчанию, и никаких дополнительных манипуляций с указателями файлов проводить не требуется. Но бывает нужно контролировать указатели вручную, чтобы иметь возможность читать и писать, начиная с произвольного места файла. Функции seekg() и tellg() позволяют устанавливать и проверять текущий указатель чтения, а функции seekp() и tellp()— выполнять те же действия для указателя записи.

Вычисление позиции

Мы уже видели пример позиционирования указателя чтения в программе DISKFUN, где функция seekg() устанавливала его на начало файла с тем, чтобы чтение начиналось оттуда. Этот вариант seekg() имеет один аргумент, представляющий собой абсолютную позицию относительно начала файла. Начало принимается за 0, что и использовалось в DISKFUN. Это условно показано на рис. 12.4

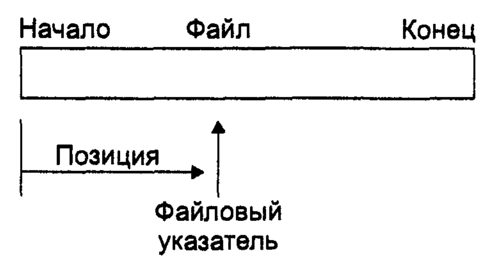


Рис. 12.4. Функция seekg() с одним аргументом

Вычисление сдвига

Функция seekg() может использоваться в двух вариантах. Первый из них мы уже видели — используется аргумент для указания позиции относительно начала файла. Но можно использовать эту функцию и с двумя аргументами, первый из которых — сдвиг относительно определенной позиции в файле, а второй определяет позицию, начиная с которой отсчитывается этот сдвиг. Второй аргумент может иметь три значения: beg означает начало файла, cur — текущую позицию указателя файла, end — это конец файла. Например, выражение

```
seekp(-10, ios::end);
```

установит указатель записи за 10 байтов до конца файла. То, как это выглядит, условно изображено на рис. 12.5

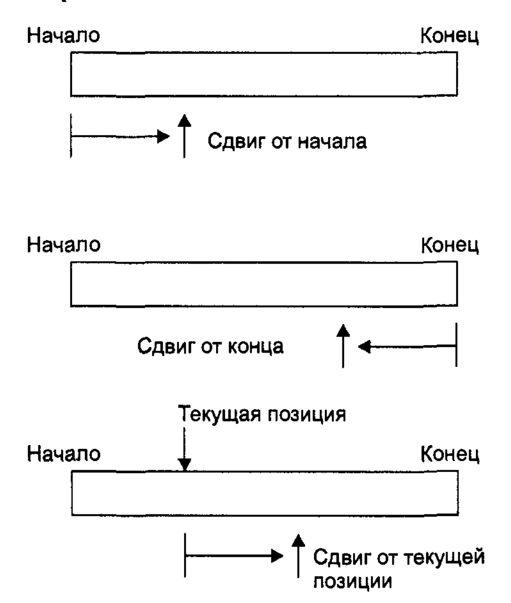


Рис. 12.5. Функция seekp() с двумя аргументами

Вот пример, в котором используется двухаргументный вариант seekg() для нахождения конкретного объекта (человека) класса person в файле GROUP.DAT и для вывода данных об этом человеке.

Листинг 12.13. Программа SEEKG

```
// seekg.cpp
// Поиск конкретного объекта в файле
#include <fstream>          // для файловых потоков
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class person                // класс person
  {
```

продолжение

Листинг 12.13 (*продолжение*)

```
  protected:
    char name[80];         // имя человека
    int age;               // его возраст
  public:
    void getData()         // получить данные о человеке
      {
      cout << "\n  Введите имя: "; cin >> name;
      cout << "  Введите возраст: "; cin >> age;
      }
    void showData(void)   // вывод данных о человеке
      {
      cout << "\n  Имя: " << name;
      cout << "\n  Возраст: " << age;
      }
  };
////////////////////////////////////////////////////////////
int main()
  {
  person pers;                    // создать объект person
  ifstream infile;                // создать входной файл
  infile.open("GROUP.DAT",ios::in|ios::binary); // открыть
                                                // файл
  infile.seekg(0, ios::end);      // установить указатель на 0
                                  // байт от конца файла
  int endposition = infile.tellg();      // найти позицию
  int n = endposition / sizeof(person);  // число человек
  cout << "\nВ файле " << n << " человек(а)";

  cout << "\nВведите номер персоны: ";
  cin >> n;
  int position = (n-1) * sizeof(person); // умножить размер
                           // данных под персону на число персон
  infile.seekg(position);                // число байт от начала
                           // прочитать одну персону
  infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
  pers.showData();                       //вывести одну персону
  cout << endl;
  return 0;
  }
```

Результат работы программы (при предположении, что файл GROUP.DAT — тот же, что в DISKFUN):

```
В файле 3 человек(а)
Введите номер персоны: 2

Имя: Артур Хайн
Возраст: 60
```

Для удобства работы пользователя мы нумеруем объекты (персоны), начиная с единицы, хотя в программе нумерация ведется с нуля. Поэтому персона №2 — это второй из трех человек, записи о которых имеются в файле.

Функция tellg()

Первое, что делает программа, это вычисляет количество человек в файле. Она делает это установкой указателя чтения на конец файла с помощью выражения

```
infile.seekg(0, ios::end);
```

Функция tellg() возвращает текущую позицию указателя чтения. Программа использует ее для вычисления длины файла в байтах. Так как длина одной записи известна, то, исходя из общей длины файла, можно узнать, сколько всего объектов хранится в файле. Результат выводится на экран.

В приведенном выше примере работы программы пользователь выбрал второй объект из файла. Программа считает с помощью seekg(), сколько нужно отступить от начала файла для того, чтобы попасть на начало области данных выбранного объекта. Затем она использует функцию read() для чтения данных, начиная с этого места. Наконец, функция showData() выводит информацию на экран.

Обработка ошибок файлового ввода/вывода

В примерах, касающихся файлового ввода/вывода, мы до сих пор не заботились о возможности возникновения ошибок и способах их обработки. В частности, мы предполагали, что открываемые для чтения файлы уже существуют, а открываемые для записи — могут быть созданы или дозаписаны. Мы также предполагали, что во время операций чтения и записи не происходило ошибок. В реальных программах, конечно, нельзя полагаться на случай, ибо если ошибка может возникнуть, то она обязательно когда-нибудь возникнет. Значит, нужны методы обработки соответствующих ситуаций. Если вы думаете, что файл существует, значит, он может и не существовать. А если считаете, что придумали уникальное имя файла, значит, файл с таким именем может и существовать, и вы перезапишете своей программой нужные данные. Вероятно, может кончиться и место на диске, гибкий диск окажется неисправным и т. д.

Реагирование на ошибки

Наша следующая программа показывает наиболее удобный способ обработки ошибок. Все дисковые операции проверяются после их выполнения. Если возникла ошибка, выводится сообщение об этом, и программа завершает свою работу. Мы использовали технику, описанную ранее, заключающуюся в проверке значения, возвращаемого из объекта, и определении статуса его ошибки. Программа открывает выходной потоковый объект, записывает целый массив целых чисел единственным вызовом write() и закрывает объект. Затем открывает входной потоковый объект и считывает массив вызовом функции read().

Листинг 12.14. Программа REWERR

```
// rewerr.cpp
// Обработка ошибок ввода/вывода
#include <fstream>           // для файловых потоков
#include <iostream>
using namespace std;
#include <process.h>         // для exit()

const int MAX = 1000;
int buff[MAX];

int main()
  {
  for(int j=0; j<MAX; j++)    // заполнить буфер данными
    buff[j] = j;
  ofstream os;               // создать выходной поток
                             // открыть его
  os.open("a:edata.dat", ios::trunc | ios::binary);
  if(!os)
    { cerr << "Невозможно открыть выходной файл\n"; exit(1); }

  cout << "Идет запись...\n"; // записать в него содержимое
                             // буфера
  os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
  if(!os)
    { cerr << "Запись в файл невозможна\n"; exit(1); }
  os.close();                // надо закрыть поток

  for(j=0; j<MAX; j++)        // очистить буфер
    buff[j] = 0;

  ifstream is;               // создать входной поток
  is.open("a:edata.dat", ios::binary);
  if(!is)
    { cerr << "Невозможно открыть входной файл\n";exit(1); }

  cout << "Идет чтение...\n"; // чтение файла
  is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
  if(!is)
    { cerr << "Невозможно чтение файла\n"; exit(1); }

  for(j=0; j<MAX; j++)        // проверять данные
    if( buff[j] != j )
      { cerr << "\nДанные некорректны\n"; exit(1); }
  cout << "Данные в порядке\n";
  return 0;
  }
```

Анализ ошибок

В этом примере мы определили наличие ошибки ввода/вывода проверкой значения, возвращаемого потоковым объектом.

```
if(!is)
//Возникла ошибка
```

Здесь is возвращает значение указателя, если все прошло без ошибок. В противном случае возвращается 0. Это жесткий подход к определению ошибок: не имеет значения, какая именно ошибка возникла, все ошибки обрабатываются одинаково. С помощью флагов статуса ошибок ios, тем не менее, можно извлечь более подробную информацию об ошибках файлового ввода/вывода.

Мы уже видели некоторые из этих флагов статуса в работе, когда обсуждали вывод на экран и ввод с клавиатуры. В следующем примере показано, как можно использовать их при файловом вводе/выводе.

Листинг 12.15. Программа FERRORS

```
// ferrors.cpp
// Проверка ошибок открытия файла
#include <fstream>          // для файловых функций
#include <iostream>
using namespace std;

int main()
  {
  ifstream file;
  file.open("a:test.dat");

  if( !file )
    cout << "\nНевозможно открыть GROUP.DAT";
  else
    cout << "\nФайл открыт без ошибок.";
  cout << "\nfile = " << file;
  cout << "\nКод ошибки = " << file.rdstate();
  cout << "\ngood() = " << file.good();
  cout << "\neof() = " << file.eof();
  cout << "\nfail() = " << file.fail();
  cout << "\nbad() = " << file.bad() << endl;
  file.close();
  return 0;
  }
```

Вначале программа проверяет значение файлового объекта. Если оно нулевое, значит, файл, возможно, не существует и поэтому не может быть открыт. Вот результат работы программы в такой ситуации:

```
Невозможно открыть GROUP.DAT
file = 0x1c730000
Код ошибки = 4
good() = 0
eof() = 0
fail() = 4
bad() = 4
```

Код ошибки, возвращаемый rdstate(), равен четырем. Это бит, который говорит о том, что файл не существует. Он устанавливается в единицу. Остальные биты при этом равны нулю. Функция good() возвращает 1 (true) лишь в том случае, когда не установлены никакие биты ошибок, поэтому в нашем примере она вернула 0 (false). Указатель файла находится не на EOF, поэтому eof() возвращает 0. Флаги fail() и bad() устанавливаются в ненулевое значение — это их реакция на произошедшую ошибку.

В настоящей, серьезной программе некоторые из этих функций должны использоваться после каждой операции ввода/вывода, и тогда не останется сомнений в том, что все идет как нужно.

Файловый ввод/вывод с помощью методов

Ранее мы сводили все функции работы с файлами в секцию main() наших программ. Но при написании сложных классов логичнее включать операции файлового ввода/вывода в методы класса. В этом параграфе мы продемонстрируем две программы, в которых это проделано. В первой из них используются обычные методы, каждый объект отвечает за запись самого себя в файл и чтение из файла. Вторая показывает, как файловый ввод/вывод всех объектов можно осуществить с помощью статических функций-членов класса.

Как объекты записывают и читают сами себя

Иногда имеет смысл разрешить каждому компоненту класса читать и записывать самого себя. Это довольно прозрачный подход и отлично работает, если нужно писать и читать небольшое число объектов. В нашем примере мы добавили два метода — diskOut() и diskIn() — в класс person. Эти функции позволяют объектам person записывать себя в файл и читать себя же из него.

Примем некоторые упрощающие допущения. Во-первых, все объекты будут храниться в одном и том же файле PERSFILE.DAT. Во-вторых, новые объекты всегда будут добавляться к концу файла. Аргумент функции diskIn() позволяет читать данные о любом человеке из файла. Для предотвращения попыток прочитать данные, выходящие за пределы файла, мы включаем в программу статический метод diskCount(), возвращающий число людей, информация о которых хранится в файле. При вводе данных следует использовать только фамилии людей, пробелы не допускаются.

Листинг 12.16. Программа REWOBJ

```
// rewobj.cpp
// Файловый ввод/вывод объектов person
#include <fstream>          // Для файловых потоков
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class person                // класс person
  {
  protected:
    char name[40];          // имя человека
    int age;                // его возраст
  public:
    void getData(void)      // получить данные
      {
      cout << "\n  Введите фамилию: "; cin >> name;
      cout << "  Введите возраст: "; cin >> age;
      }
    void showData(void)     // Вывод данных
```

```
    {
    cout << "\n  Имя: " << name;
    cout << "\n  Возраст: " << age;
    }
  void diskIn(int);          // чтение из файла
  void diskOut();            // запись в файл
  static int diskCount();    // Число человек в файле
};
//----------------------------------------------------------
void person::diskIn(int pn)  // Чтение данных о числе
                             // человек pn из файла
{
ifstream infile;             // создать поток
infile.open("PERSFILE.DAT", ios::binary);  // открыть его
infile.seekg( pn*sizeof(person) );         // сдвиг
                                           // файлового указателя
infile.read( (char*)this, sizeof(*this) ); // чтение данных
                                           // об одном человеке
}
//----------------------------------------------------------
void person::diskOut()       // запись в конец файла
{
ofstream outfile;            // создать поток
                             // открыть его
outfile.open("PERSFILE.DAT", ios::app | ios::binary);
outfile.write((char*)this,sizeof(*this)); //записать в него
}
//----------------------------------------------------------
int person::diskCount()      // число людей в файле
{
ifstream infile;
infile.open("PERSFILE.DAT", ios::binary);
infile.seekg(0, ios::end);   // перейти на позицию «0 байт
                             // от конца файла»
                             // вычислить количество людей
return (int)infile.tellg() / sizeof(person);
}
////////////////////////////////////////////////////////////
int main()
{
person p;                    // создать пустую запись
char ch;

do {                         // сохранение данных на диск
  cout << "Введите данные о человеке: ";
  p.getData();               // Получить данные
  p.diskOut();               // записать на диск
  cout << "Продолжить (y/n)? ";
  cin >> ch;
  } while(ch=='y');          // цикл до 'n'

int n = person::diskCount(); // сколько людей в файле?
cout << "В файле " << n << " человек(а)\n";
for(int j=0; j<n; j++)       // для каждого
  {
  cout << "\nПерсона " << j;
  p.diskIn(j);               // считать с диска
  p.showData();              // вывести данные
```

продолжение

Листинг 12.16 (*продолжение*)

```
  }
 cout << endl;
 return 0;
 }
```

Здесь для вас не должно быть новых откровений. Большинство элементов этой программы уже встречались. Она работает примерно по тому же принципу, что и DISKFUN. Тем не менее имейте в виду, что все подробности дисковых операций невидимы для main(), они спрятаны внутрь класса person.

Заранее никогда неизвестно, где находятся данные, с которыми мы собираемся работать, так как каждый объект находится в своей области памяти. Но указатель this всегда подскажет, где мы находимся во время выполнения какого-либо метода. В потоковых функциях read() и write() адрес объекта, который будет читаться или записываться, равен *this, а его размер — sizeof(*this).

Вот результат работы программы при предположении, что до начала ее работы в файле уже было две записи:

```
Введите данные о человеке:
Введите имя: Гребеньков
Введите возраст:19
Продолжить (y/n)? y

Введите данные о человеке:
Введите имя: Андреанов
Введите возраст:20
Продолжить (y/n)? n

Персона #1
Имя:Ершова
Возраст:21
Персона #2
Имя:Малахова
Возраст:20
Персона #3
Имя:Гребеньков
Возраст:19
Персона #4
Имя:Андреанов
Возраст:20
```

Чтобы пользователь мог ввести собственное имя файла, вместо жесткого закрепления его в программе, как мы делали в предыдущих примерах, следует создать статическую компонентную переменную (например, char fileName[]), а также статическую функцию для ее установки. Можно сделать и по-другому: каждый объект записывать в свой файл. Для этого понадобится нестатическая функция.

Как классы записывают и читают сами себя

Предположим, что в памяти хранится большое число объектов, и все их нужно записать в файлы. Недостаточно иметь для каждого из них метод, который откроет файл, запишет в него объект, потом закроет файл, как было в предыдущем

примере. Гораздо быстрее и логичнее в такой ситуации открыть файл, записать в него все объекты, которые этого хотят, потом закрыть файл.

Статические функции

Одним из способов записать за один сеанс множество объектов является употребление статической функции, которая применяется ко всему классу в целом, а не к отдельным его объектам. Такая функция действительно может записать все объекты сразу. Но как она узнает, где находятся объекты? Она обратится к массиву указателей на объекты, который можно хранить в виде статической переменной. При создании каждого объекта его указатель заносится в этот массив. Статический элемент данных также может свято хранить память о том, сколько всего объектов было создано. Статическая функция write() откроет файл, в цикле пройдет по всем ссылкам массива, записывая по очереди все объекты, и после окончания записи закроет файл.

Размеры порожденных объектов

Чтобы жизнь медом не казалась, давайте сделаем следующее предположение: объекты, хранящиеся в памяти, имеют разные размеры. Почему и когда такое бывает? Типичной предпосылкой этого является создание порожденных классов. Например, рассмотрим программу EMPLOY из главы 9 «Наследование». Там у нас был класс employee, выступавший как базовый для классов manager, scientist и laborer. Объекты этих трех порожденных классов имеют разные размеры, так как содержат разные объемы данных. Например, в дополнение к имени и порядковому номеру, которые присущи всем работникам, у менеджера есть еще такие поля данных, как должность и членские взносы гольф-клуба, а у ученого есть поле данных, содержащее количество его публикаций.

Нам хотелось бы записать данные из списка, содержащего все три типа порожденных объектов, используя простой цикл и метод write() класса ofstream. Но нам нужно знать размеры объектов, чтобы передать их в качестве второго аргумента этой функции.

Пусть имеется массив указателей arrap[], хранящий ссылки на объекты типа employee. Указатели могут ссылаться, таким образом, и на все три порожденных класса (см. программу VIRTPERS из главы 11, там тоже имеется массив указателей на объекты порожденных классов). При использовании виртуальных функций можно использовать выражения типа

```
arrap[]->putdata();
```

Тогда будет поставлена на исполнение во время работы программы версия putdata(), соответствующая объекту, на который ссылается указатель, а не та, что соответствует базовому классу. Можно ли использовать sizeof() для вычисления размера ссылочного аргумента? То есть можно ли написать, например, такое выражение:

```
ouf.write( (char*)arrap[j], sizeof(*arrap[j]) ); //плохо это
```

Увы, нельзя, потому как sizeof() не является виртуальной функцией. Она не в курсе, что нужно обращаться к типу объекта, на который ссылается указатель,

а не на тип самого указателя. Эта функция всегда будет возвращать размер объекта базового класса.

Использование функции typeid()

И все-таки, как же нам найти размер объекта, если все, что мы имеем, это указатель на него? Одним из решений является, несомненно, функция typeid(), представленная в главе 11. Можно использовать ее для определения класса объекта, подставляя имя этого класса в качестве аргумента sizeof(). Чтобы использовать typeid(), надо включить параметр компилятора *RTTI.* (это существенно только для компилятора Microsoft Visual C++, см. приложение В «Microsoft Visual C++»).

Наш следующий пример показывает, как все вышеописанное работает. Узнав размер объекта, мы можем использовать его в функции write() для записи в файл.

К программе EMPLOY добавился несложный пользовательский интерфейс. Некоторые специфические методы сделаны виртуальными, чтобы можно было пользоваться массивом указателей на объекты. В программу также включены некоторые приемы обнаружения ошибок, описанные в предыдущем параграфе.

Программа получилась довольно претенциозной, но, тем не менее, она действительно демонстрирует многие приемы, которые могут быть использованы в настоящем приложении для работы с базами данных. Она также показывает реальные возможности ООП. Что бы мы делали без него? Как бы смогли с помощью одного выражения записывать в файл объекты разных размеров? Приводим листинг программы EMPL_IO и рекомендуем вникнуть в каждую строчку кода.

Листинг 12.17. Программа EMPL_IO

```
// empl_io.cpp
// Файловый ввод/вывод объектов employee
// Поддержка объектов неодинаковых размеров
#include <fstream>      // для потоковых файловых функций
#include <iostream>
#include <typeinfo>     // для typeid()
using namespace std;
#include <process.h>    // для exit()

const int LEN = 32;     // Максимальная длина фамилий
const int MAXEM = 100;  // максимальное число работников

enum employee_type {tmanager, tscientist, tlaborer};
//////////////////////////////////////////////////////////////
class employee                  // класс employee
  {
  private:
    char name[LEN];             // фамилия работника
    unsigned long number;       // номер работника
    static int n;               // текущее число работников
    static employee* arrap[];   //массив указателей на
                                // класс работников
  public:
    virtual void getdata()
      {
      cin.ignore(10, '\n');
      cout << "  Введите фамилию: "; cin >> name;
```

```
      cout << "  Введите номер: ";    cin >> number;
      }
    virtual void putdata()
      {
      cout << "\n  Фамилия: " << name;
      cout << "\n  Номер: " << number;
      }
    virtual employee_type get_type(); // получить тип
    static void add();                // добавить работника
    static void display();            // вывести данные обо всех
    static void read();               // чтение из файла
    static void write();              // запись в файл
  };
//---------------------------------------------------------
//статические переменные
int employee::n;                  // текущее число работников
employee* employee::arrap[MAXEM]; // массив указателей на
                                  // класс работников
///////////////////////////////////////////////////////////
//класс manager (менеджеры)
class manager : public employee
  {
  private:
    char title[LEN];  // титул ("вице-президент" и т. п.)
    double dues;      // Налоги гольф-клуба
  public:
    void getdata()
      {
      employee::getdata();
      cout << "  Введите титул: ";       cin >> title;
      cout << "  Введите налоги: "; cin >> dues;
      }
    void putdata()
      {
      employee::putdata();
      cout << "\n  Титул: " << title;
      cout << "\n  Налоги гольф-клуба: " << dues;
      }
  };
///////////////////////////////////////////////////////////
//класс scientist (ученые)
class scientist : public employee
  {
  private:
    int pubs;              // число публикаций
  public:
    void getdata()
      {
      employee::getdata();
      cout << "  Введите число публикаций: "; cin >> pubs;
      }
    void putdata()
      {
      employee::putdata();
      cout << "\n  Число публикаций: " << pubs;
      }
  };
```

продолжение

Листинг 12.17 (*продолжение*)

```
////////////////////////////////////////////////////////////
//класс laborer (рабочие)
class laborer : public employee
  {
  };
////////////////////////////////////////////////////////////
//добавить работника в список (хранится в ОП)
void employee::add()
  {
  char ch;
  cout << "'m' для добавления менеджера"
        "\n's' для добавления ученого"
        "\n'l' для добавления рабочего"
        "\nВаш выбор: ";
  cin >> ch;
  switch(ch)
    {          //создать объект указанного типа
    case 'm': arrap[n] = new manager;  break;
    case 's': arrap[n] = new scientist; break;
    case 'l': arrap[n] = new laborer;  break;
    default: cout << "\nНеизвестный тип работника\n"; return;
    }
  arrap[n++]->getdata();  //Получить данные от пользователя
  }
//----------------------------------------------------------
//Вывести данные обо всех работниках
void employee::display()
  {
  for(int j=0; j<n; j++)
    {
    cout  << (j+1);               // вывести номер
    switch( arrap[j]->get_type() ) //вывести тип
      {
      case tmanager:  cout << ". Тип: Менеджер";  break;
      case tscientist: cout << ". Тип: Ученый"; break;
      case tlaborer:   cout << ". Тип: Рабочий";  break;
      default: cout << ". Неизвестный тип";
      }
    arrap[j]->putdata();          // Вывод данных
    cout << endl;
    }
  }
//----------------------------------------------------------
//Возврат типа объекта
employee_type employee::get_type()
  {
  if( typeid(*this) == typeid(manager) )
    return tmanager;
  else if( typeid(*this)==typeid(scientist) )
    return tscientist;
  else if( typeid(*this)==typeid(laborer) )
    return tlaborer;
  else
    { cerr << "\nНеправильный тип работника"; exit(1); }
  return tmanager;
  }
```

```
//--------------------------------------------------------
//Записать все объекты, хранящиеся в памяти, в файл
void employee::write()
  {
  int size;
  cout << "Идет запись " << n << " работников.\n";
  ofstream ouf;           // открыть ofstream в двоичном виде
  employee_type etype;   // тип каждого объекта employee

  ouf.open("EMPLOY.DAT", ios::trunc | ios::binary);
  if(!ouf)
    { cout << "\nНевозможно открыть файл\n"; return; }
  for(int j=0; j<n; j++)  // Для каждого объекта
    {                     // получить его тип
    etype = arrap[j]->get_type();
                          // записать данные в файл
    ouf.write( (char*)&etype, sizeof(etype) );
    switch(etype)         // find its size
      {
      case tmanager:  size=sizeof(manager); break;
      case tscientist: size=sizeof(scientist); break;
      case tlaborer:  size=sizeof(laborer); break;
      }      //запись объекта employee в файл   ouf.write( (char*)(arrap[j]), size );
    if(!ouf)
      { cout << "\nЗапись в файл невозможна\n"; return; }
    }
  }
//--------------------------------------------------------
//чтение всех данных из файла в память
void employee::read()
  {
  int size;               // размер объекта employee
  employee_type etype;    // тип работника
  ifstream inf;           // открыть ifstream в двоичном виде
  inf.open("EMPLOY.DAT", ios::binary);
  if(!inf)
    { cout << "\nНевозможно открыть файл\n"; return; }
  n = 0;                  // В памяти работников нет
  while(true)
    {                     // чтение типа следующего работника
    inf.read( (char*)&etype, sizeof(etype) );
    if( inf.eof() )       // выход из цикла по EOF
      break;
    if(!inf)              // ошибка чтения типа
      { cout << "\nНевозможно чтение типа\n"; return; }
    switch(etype)
      {                   // создать нового работника
      case tmanager:      // корректного типа
        arrap[n] = new manager;
        size=sizeof(manager);
        break;
      case tscientist:
        arrap[n] = new scientist;
        size=sizeof(scientist);
        break;
```

продолжение

Листинг 12.17 (*продолжение*)

```
      case tlaborer:
        arrap[n] = new laborer;
        size=sizeof(laborer);
        break;
      default: cout << "\nНеизвестный тип в файле\n"; return;
      }                    // чтение данных из файла
    inf.read( (char*)arrap[n], size  );
    if(!inf)               // ошибка, но не EOF
      { cout << "\nЧтение данных из файла невозможно\n"; return; }
    n++;                   // счетчик работников увеличить
    } //end while
  cout << "Идет чтение " << n << " работников\n";
  }
///////////////////////////////////////////////////////////
int main()
  {
  char ch;
  while(true)
    {
    cout << "'a' - добавление сведений о работнике"
         "\n'd' - вывести сведения обо всех работниках"
         "\n'w' - записать все данные в файл"
         "\n'r' - прочитать все данные из файла
         "\n'x' - выход"
         "\nВаш выбор: ";
    cin >> ch;
    switch(ch)
      {
      case 'a':            // добавить работника
        employee::add(); break;
      case 'd':            // вывести все сведения
        employee::display(); break;
      case 'w':            // запись в файл
        employee::write(); break;
      case 'r':            // чтение всех данных из файла
        employee::read(); break;
      case 'x': exit(0);   // выход
      default: cout << "\nНеизвестная команда";
      } //end switch
    } //end while
  return 0;
  } //end main()
```

Код типа объекта

Мы умеем определять класс объекта, находящегося в памяти, но как узнать класс объекта, если мы собираемся читать его из файла? Никакой волшебной палочки в виде специальной функции на этот случай не предусмотрено. Поэтому при записи данных на диск необходимо записывать код (перечисляемую переменную типа employee_type) прямо перед данными объекта. До начала чтения из файла надо прочитать его значение и создать объект соответствующего типа. И только после этого можно копировать данные из файла в этот новый объект.

И никаких кустарных объектов, пожалуйста!

В голову иного программиста может прийти идея считывать данные объекта просто «куда-нибудь», например в массив типа char, а затем устанавливать указатель на этот массив.

```
char someArray[MAX];
aClass* aPtr_to_Obj;
        //Никогда так не делайте!
aPtr_to_Obj = reinterpret_Cast<aClass*>(someArray);
```

Такой сделанный на коленке объект не будет создан. И попытки работать с указателем, делая вид, что он ссылается на него, ни к чему хорошему не приведут. Есть только два легитимных пути создать объект. Можно определить его вручную, тогда он будет создан на этапе компиляции:

```
aClass anObj;
```

Либо, если вы хотите, чтобы объект создавался в процессе работы программы, используйте new и ассоциируйте его адрес с указателем:

```
aPtr_to_Obj = new aClass;
```

При корректном способе создания объекта запускается его конструктор. Это необходимо даже тогда, когда вы не определяете собственный, а используете конструктор по умолчанию. Ведь объект — это нечто более глобальное, нежели просто область памяти, в которой хранятся данные. Не забывайте, что это еще и набор методов, некоторые из них вы даже не видите!

Взаимодействие с программой EMPL_IO

Вот пример взаимодействия пользователя с нашей программой. Напоминаем, что в памяти у нас созданы объекты классов manager, scientist и laborer. Мы записываем их на диск, затем считываем и выводим на экран. Для простоты названия из нескольких слов мы использовать не будем.

```
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: m
Введите фамилию: Александров
Введите номер:1111
Введите титул:Президент
Введите налоги:20000

'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
```

```
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: s
Введите фамилию: Лебедев
Введите номер:2222
Введите число публикаций: 99

'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: a
'm' для добавления менеджера
's' для добавления ученого
'l' для добавления рабочего
Ваш выбор: l
Введите фамилию: Шевелев
Введите номер:3333

'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: w
Идет запись 3 работников

'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: r
Идет чтение 3 работников
'a' - добавление сведений о работнике
'd' - вывести сведения обо всех работниках
'w' - записать все данные в файл
'r' - прочитать все данные из файла
'x' - выход
Ваш выбор: d
1.Тип:Менеджер
Фамилия: Александров
Номер:1111
Титул:Президент
Налоги гольф-клуба: 20000
2.Тип:Ученый
Фамилия: Лебедев
Номер:2222
Число публикаций: 99
3.Тип:рабочий
Фамилия:Шевелев
Номер:3333
```

Конечно же, можно выйти из программы и после записи на диск. Когда вы повторно запустите программу, все данные снова появятся и смогут быть прочитаны.

Эту программу легко расширить за счет добавления функций удаления работника, извлечения данных об одном конкретном работнике, поиска работника с конкретными характеристиками и т. д.

Перегрузка операторов извлечения и вставки

Давайте перейдем к изучению следующей темы, связанной с потоками. Данный раздел будет посвящен перегрузке операторов извлечения и вставки. Это мощный прием C++, который позволяет поддерживать ввод/вывод пользовательских типов наравне со стандартными, такими, как float и int. Например, если имеется объект класса crawdad, называющийся cd1, его можно вывести на экран простым выражением

```
cout << "\ncd1=" << cd1;
```

Как видите, оно ничем не отличается от подобных конструкций для стандартных типов.

Операторы извлечения и вставки могут быть перегружены и для работы с консольным вводом/выводом (с клавиатуры и на экран — cin и cout). С оглядкой, но можно перегрузить их и для работы с дисковыми файлами. Мы рассмотрим примеры всех этих ситуаций.

Перегрузка cout и cin

Приведем пример, в котором операторы извлечения и вставки для класса Distance перегружены для работы с cout и cin.

Листинг 12.18. Программа ENGLIO

```
// englio.cpp
// Перегружаемые операции << и >>
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                          // класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
    Distance() :feet(0),inches(0.0) // конструктор (0 арг.)
      { }
                                    // конструктор (2 аргумента)
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
```

продолжение

Листинг 12.18 (*продолжение*)

```
    friend istream& operator >> (istream& s, Distance& d);
    friend ostream& operator << (ostream& s, Distance& d);
  };
//--------------------------------------------------------------
istream& operator >> (istream& s, Distance& d) //получить
                               // значение от пользователя
  {
  cout << "\nВведите футы: ";  s >> d.feet;  // используется
  cout << "Введите дюймы: ";  s >> d.inches; // перегруженный
  return s;                                  // оператор >>
  }
//--------------------------------------------------------------
ostream& operator << (ostream& s, Distance& d)       // вывести
  {                                          // расстояние
  s << d.feet << "\'-" << d.inches << '\"'; // используется
  return s;                                  // перегруженный <<
  }
////////////////////////////////////////////////////////////////
int main()
  {
  Distance dist1, dist2;     //Определение переменных
  Distance dist3(11, 6.25);  // определение, инициализация dist3
  cout << "\nВведите два значения расстояний:";
  cin >> dist1 >> dist2;     // Получить значения от пользователя
                             // вывод расстояний
  cout << "\ndist1 = " << dist1 << "\ndist2 = " << dist2;
  cout << "\ndist3 = " << dist3 << endl;
  return 0;
  }
```

Программа запрашивает у пользователя два значения расстояний (типа Distance), выводит их, а также выводит значения, которые были инициализированы в программе.

```
Введите футы:10
Введите дюймы:3.5

Введите футы:12
Введите дюймы:6
dist1 = 10' -3.5"
dist2 = 12' -6"
dist3= 11' -6.25"
```

Обратите внимание, как удобно и естественно можно работать с объектами типа Distance, — совершенно так же, как с объектами любого другого типа:

```
cin >> dist1 >> dist2;
```

Или, например,

```
cout << "\ndist1=" << dist1 << "\ndist2=" << dist2;
```

Операторы извлечения и вставки перегружаются одинаковыми способами. Они возвращают по ссылке объект istream (для >>) или ostream (для <<). Возвращаемые значения могут организовываться в цепочки. Операторы имеют по два аргумента, оба передаются по ссылке. Первый аргумент >> — объект класса istream (например, cin). Соответственно, для << первым аргументом должен быть

объект класса ostream (например, cout). Второй аргумент — объект класса, для которого осуществляется ввод/вывод (в данном примере это класс Distance). Оператор >> берет входные данные из потока, указанного в первом аргументе, и переносит их в компонентные данные объекта, указанного во втором. По аналогии оператор << берет данные из объекта, указанного во втором аргументе, и посылает их в поток, соответствующий значению первого аргумента.

Функции operator<<() и operator>>() должны быть дружественными по отношению к классу Distance, так как объекты istream и ostream находятся слева от знака операции (см. обсуждение дружественных функций в главе 11).

Разумеется, для любых других классов можно перегружать операторы вставки и извлечения таким же способом.

Перегрузка << и >> для файлов

Наш следующий пример продемонстрирует, как перегружаются операторы << и >> в классе Distance для работы с файловым вводом/выводом.

Листинг 12.19. Программа ENGLIO2

```
// englio2.cpp
// перегружаемые операции << и >> могут работать с файлами
#include <fstream>
#include <iostream>
using namespace std;
class Distance              // класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
    Distance() : feet(0), inches(0.0) // конструктор (без
                                      // аргументов)
      { }                   // конструктор (два аргумента)
    Distance(int ft, float in) : feet(ft), inches(in)
      { }
    friend istream& operator >> (istream& s, Distance& d);
    friend ostream& operator << (ostream& s, Distance& d);
  };
//--------------------------------------------------------
istream& operator >> (istream& s, Distance& d) //получить
                                               // данные
  {                         // из файла или с клавиатуры
  char dummy;               // для ('), (-) и (")
                            //с помощью перегруженного >>
  s >> d.feet >> dummy >> dummy >> d.inches >> dummy;
  return s;
  }
//--------------------------------------------------------
ostream& operator << (ostream& s, Distance& d) // послать
                            // данные типа Distance
  {                         // в файл или
  s << d.feet << "\'-" << d.inches << '\"';  // на экран
  return s;                 // перегруженным <<
  }
```

продолжение

Листинг 12.19 (*продолжение*)

```
////////////////////////////////////////////////////////////
int main()
  {
  char ch;
  Distance dist1;
  ofstream ofile;          // создать и открыть
  ofile.open("DIST.DAT"); // выходной поток

  do {
    cout << "\nВведите расстояние: ";
    cin >> dist1;          // получить данные от пользователя
    ofile << dist1;        // записать их в выходной поток
    cout << "Продолжать (y/n)? ";
    cin >> ch;
    } while(ch != 'n');
  ofile.close();           // закрыть выходной поток

  ifstream ifile;          // создать и открыть
  ifile.open("DIST.DAT"); // входной поток

  cout << "\nСодержимое файла:\n";
  while(true)
    {
    ifile >> dist1;        // чтение данных из потока
    if( ifile.eof() )      // выход по EOF
      break;
    cout << "Расстояние = " << dist1 <<endl; //вывод
                                              // расстояний
    }
  return 0;
  }
```

В сами перегружаемые операции мы внесли лишь минимальные изменения. Оператор >> больше не просит ввести входные данные, потому что понимает, что бессмысленно просить о чем-либо файл. Мы предполагаем, что пользователь знает, как точно вводить значения футов и дюймов, включая знаки пунктуации. Оператор << остался без изменений. Программа запрашивает у пользователя входные данные и после получения каждого значения записывает его в файл. Когда пользователь оканчивает ввод данных, программа читает из файла и выводит на экран все хранящиеся там значения. Вот пример работы программы:

```
Введите расстояние: 3'-4.5"
Продолжать (y/n)? y

Введите расстояние: 7'-11.25"
Продолжать (y/n)? y

Введите расстояние: 11'-6"
Продолжать (y/n)? n
Содержимое файла:
Расстояние = 3'-4.5"
Расстояние = 7'-11.25"
Расстояние = 11'-6"
```

Значения расстояний записаны в файле символ за символом. Поэтому реальное содержание файла таково:

```
3'-4.5"7'-11.25"11'-6"
```

Если пользователь ошибется при вводе данных, они не будут записаны корректно в файл и не смогут быть прочитаны оператором <<. В реальной программе необходимо проверять, правильно ли производится ввод.

Память как поток

Область памяти можно считать потоком и записывать в нее данные точно так же, как в файл. Это требуется, когда нужно выводить данные в определенном формате (например, оставлять только два знака после запятой) и одновременно использовать функцию текстового вывода, которая в качестве аргумента требует строку. Так обычно делают, вызывая функции вывода, при разработке приложений в GUI-средах, например в Windows, поскольку там этим функциям нужно передавать именно строку в качестве аргумента. Программисты на C, вероятно, сразу вспомнят, как они использовали sprintf() для этих целей.

Семейство потоковых классов поддерживает такое форматирование данных в памяти. Для вывода в память существует специальный класс ostrstream, порожденный классом ostream. Для ввода из памяти служит класс istrstream, порожденный, соответственно, классом istream. Для объектов, которым требуется осуществлять одновременно ввод и вывод, создан класс strstream — наследник iostream.

Скорее всего, вы захотите использовать ostrstream. Наш следующий пример показывает, как это реально применить на практике. Начните с создания буфера данных в памяти. Затем сваяйте объект ostrstream, используя этот буфер (его адрес и размер) в качестве аргументов конструктора потока. Теперь можно выводить форматированный текст в буфер, как если бы он был потоковым объектом.

Листинг 12.20. Программа OSTRSTR

```
// ostrstr.cpp
// Запись форматированных данных в память
#include <strstream>
#include <iostream>
#include <iomanip>          // для функции setiosflags()
using namespace std;
const int SIZE = 80;        // размер буфера

int main()
  {
  char ch = 'x';            // тестовые данные
  int j = 77;
  double d = 67890.12345;
  char str1[] = "Kafka";
  char str2[] = "Freud";

  char membuff[SIZE];               // буфер в памяти
  ostrstream omem(membuff, SIZE); // создать потоковый объект
```

продолжение ⇨

Листинг 12.20 (*продолжение*)
```
omem << "ch=" << ch << endl      // вставить форматированные
                                 // данные
     << "j=" << j << endl        // в объект
     << setiosflags(ios::fixed)  // формат с десятичной
                                 // запятой (фиксированной)
     << setprecision(2)          // оставлять два знака после
                                 // запятой
     << "d=" << d << endl
     << "str1=" << str1 << endl
     << "str2=" << str2 << endl
     << ends;          // закончить буфер символом '\0'
cout << membuff;       // вывод содержимого буфера
return 0;
}
```

После запуска программы membuff заполнится форматированным текстом:

```
ch=x\nj=77\nd=67890.12\nstr1=Kafka\nstr2=Freud\n\0
```

Можно форматировать данные с плавающей запятой привычными способами. Здесь же мы используем формат с фиксированной запятой (не экспоненциальный), указав на это с помощью ios::fixed, при этом оставляем два знака после запятой. Манипулятор ends вставляет в конец строки служебный символ '\0' для создания признака EOF. Вывод на экран содержимого буфера с использованием обычного cout дает такой результат:

```
ch=x
j=77
d=67890.12
str1=Kafka
str2=Freud
```

Здесь выводится содержимое буфера просто для демонстрации того, что кое-что из написанного в нашей программе работает. Вообще же, конечно, такие методы используются для более сложных операций с данными.

Аргументы командной строки

Если вы использовали когда-нибудь старый добрый MS DOS, вам должно быть знакомо понятие аргументов командной строки, использующихся при запуске программ. Их типичное применение — передача имени файла с данными в приложение. Например, вы запускаете программу редактирования текстов и хотите сразу открыть документ, с которым будете работать. Для этого вы пишете:

```
C:\>wordproc afile.doc
```

Здесь аргументом командной строки является afile.doc. Нам надо каким-то образом заставить программу на C++ распознавать такие обращения. Следующая программа, приводимая в качестве примера, считывает и выводит на экран столько аргументов командной строки, сколько вы сможете напечатать (они отделяются пробелами).

Листинг 12.21. Программа COMLINE

```
// comline.cpp
// Демонстрация работы с аргументами командной строки
#include <iostream>
using namespace std;

int main(int argc, char* argv[] )
  {
  cout << "\nargc = " << argc << endl; // число аргументов

  for(int j=0; j<argc; j++)          // вывести аргументы
    cout << "Аргумент " << j << " = " << argv[j] << endl;
  return 0;
  }
```

А вот пример работы этой программы:

```
C:\C++BOOK\Chap12>comline uno dos tres
argc = 4
Аргумент 0 = C:\C++BOOK\CHAP12>COMLINE.exe
Аргумент 1 = uno
Аргумент 2 = dos
Аргумент 3 = tres
```

Для того чтобы программа смогла прочитать аргументы, функции main() (не забываем, что это тоже функция!) должны быть самой переданы два аргумента. Первый, argc (*счетчик аргументов*), представляет собой число параметров, переданных из командной строки в программу. Первый параметр — это всегда полный путь к данной программе. Остальные уже зависят от того, что ввел пользователь. Друг от друга аргументы отделяются пробелом. В приведенном примере аргументами командной строки были uno, dos, tres.

Система хранит переданные ей параметры как строки в памяти. Создает массив указателей на эти строки. В нашем примере этим массивом был argv. Каждая отдельная строчка доступна с помощью соответствующего указателя, поэтому первым аргументом (путем к программе) является argv[0], вторым — argv[1] и т. д. COMLINE обращается по очереди к аргументам и выводит их на экран с помощью цикла for, использующего в качестве ограничителя параметр argc (число переданных в программу аргументов).

Нет особой нужды использовать только имена argv и argc в качестве параметров main(), но они настолько привычны, что другие названия вызовут у всех, кроме невозмутимого компилятора, повышенное напряжение мозговой деятельности в попытках понять, чего хотел программист.

Приводимая ниже программа использует аргумент командной строки для чего-то действительно полезного. Она выводит на экран содержимое текстового файла, имя которого пользователь ввел в командной строке. Таким образом, она имитирует команду TYPE MS DOS.

Листинг 12.22. Программа OTYPE

```
// otype.cpp
// Имитация команды TYPE
#include <fstream>          // для файловых функций
#include <iostream>
```

продолжение

Листинг 12.22 (*продолжение*)

```
using namespace std;
#include <process.h>       // для exit()

int main(int argc, char* argv[] )
  {
  if( argc != 2 )
    {
    cerr << "\nФормат команды: otype имя_файла";
    exit(-1);
    }
  char ch;                  // символ для считывания
  ifstream infile;          // создать входной файл
  infile.open( argv[1] ); // открыть файл
  if( !infile )             // проверить на наличие ошибок
    {
    cerr << "\nНевозможно открыть " << argv[1];
    exit(-1);
    }
  while( infile.get(ch) != 0 ) // считать символ
    cout << ch;                // отобразить символ
  return 0;
  }
```

Программа вначале проверяет, правильное ли количество аргументов ввел пользователь. Надо при этом помнить, что путь к файлу OTYPE.EXE всегда будет первым аргументом. Вторым — путь к файлу, который пользователь собирается открыть.

```
C:\>otype ichar.cpp
```

Таким образом, всего аргументов должно быть ровно *два*. Если это не так, вероятно, пользователь еще не понял, как работать с нашей великой программой. Чтобы прояснить для него ситуацию, на экран выводится специальное сообщение с помощью cerr.

Если количество аргументов сошлось с требуемым, программа начинает пытаться открыть указанный файл (argv[1]). Опять же, если с этим возникли какие-то проблемы, программа выводит соответствующее сообщение. Наконец, после корректного открытия, в цикле while считывается символ за символом весь файл и выводится на экран.

Нулевое значение символа говорит о том, что достигнут конец файла. Это — еще один способ распознавания EOF. Так же можно использовать значение самого файлового объекта, как мы делали ранее:

```
while( infile )
  {
  infile.get(ch);
  cout << ch;
  }
```

А вы знаете, что весь этот цикл while можно заменить одним-единственным выражением?

```
cout << infile.rdbuf();
```

Между тем такой способ мы уже видели в программе ICHAR2.

Вывод на печатающее устройство

Не составляет никаких проблем использовать консольные программы для того, чтобы посылать данные на принтер. Операционная система создает ряд специальных имен файлов, которые обозначают различные устройства. Тем самым делается возможной работа с устройствами как с файлами. Таблица 12.11 содержит все зарезервированные под устройства имена файлов.

Таблица 12.11. Имена устройств

Имя	Устройство
con	Консоль (клавиатура и монитор)
aux или com1	Первый последовательный порт
com2	Второй последовательный порт
prn или lpt1	Первый параллельный порт
lpt2	Второй параллельный порт
lpt3	Третий параллельный порт
nul	Фиктивное (несуществующее) устройство

В большинстве систем принтер подключен к первому параллельному порту, поэтому имя принтера — prn или lpt1 (понятно, что в случае, если ваша система настроена иначе, надо использовать другое имя).

Следующая программа посылает строку и число на принтер, используя форматированный вывод (оператор вставки).

Листинг 12.23. Программа EZPRINT

```
// ezprint.cpp
// Простой вывод на принтер
#include <fstream>              // Для файловых потоков
using namespace std;

int main()
  {
  char* s1 = "\nСегодня ваше счастливое число -- ";
  int  n1 = 17982;

  ofstream outfile;             // создать выходной файл
  outfile.open("PRN");          // открыть принтеру доступ к нему
  outfile << s1 << n1 << endl;  // послать данные на принтер
  outfile << '\x0C';            // прогнать лист до конца
  return 0;
  }
```

Таким способом на принтер можно послать сколько угодно строк. Служебный символ '\x0C' осуществляет прогон страницы.

Следующая программа распечатает для вас содержимое дискового файла на принтере. В ней используется посимвольный подход к передаче данных.

Листинг 12.24. Программа OPRINT

```
// oprint.cpp
// имитация команды print
```

продолжение ☞

Листинг 12.24 (*продолжение*)

```
#include <fstream>        // для файловых функций
#include <iostream>
using namespace std;
#include <process.h>      // для exit()

int main(int argc, char* argv[] )
  {
  if(argc != 2)
    {
    cerr << "\nФормат команды: oprint имя_файла\n";
    exit(-1);
    }
  char ch;                 // символ для считывания
  ifstream infile;         // создать входной файл
  infile.open( argv[1] );  // открыть файл
  if( !infile )            // проверить на наличие ошибок
    {
    cerr << "\nНевозможно открыть " << argv[1] << endl;
    exit(-1);
    }
  ofstream outfile;        // Создать файл
  outfile.open("PRN");     // открыть доступ принтера к нему
  while( infile.get(ch) != 0 ) //считать символ
    outfile.put(ch);           // отправить символ на печать
  outfile.put('\x0C');         // прогон страницы
  return 0;
  }
```

Эта программа может быть использована для печати любых текстовых файлов, например исходных текстов программ .cpp. Она очень похожа на команду print из операционной системы MS DOS. Как и предыдущие, программа проверяет корректность количества аргументов и правильность открытия файла.

Резюме

В этой главе мы ознакомились с иерархией потоковых классов и показали, как обрабатывать различного рода ошибки ввода/вывода. Затем мы рассмотрели некоторые варианты файлового ввода/вывода. Файлы в C++ связаны с объектами различных классов: класс ofstream используется для файлового вывода, ifstream — для ввода, fstream — для ввода и вывода одновременно. Методы этих или базовых классов предназначены для выполнения операций ввода/вывода. Такие операции и функции, как <<, put() и write(), используются для вывода, а >>, get() и read() — для ввода.

Функции read() и write работают с данными в двоичном режиме. Поэтому можно записывать в файлы объекты целиком, вне зависимости от типов данных, которые они содержат. Могут храниться как отдельные объекты, так и массивы и другие структуры, составленные из множества объектов. Файловый ввод/вывод может обрабатываться с использованием методов. За него могут отвечать как конкретные объекты, так и классы (с помощью статических функций).

Проверка на наличие ошибок должна осуществляться после выполнения каждой файловой операции. Сам файловый объект принимает нулевое значение, если возникает какая-либо ошибка. К тому же для определения некоторых видов ошибок используются методы классов. Операции извлечения (>>) и вставки (<<) перегружаются для работы с пользовательскими типами данных. Память может представляться в виде потока, а данные в нее могут посылаться так же, как если бы это был файл.

Вопросы

Ответы на данные вопросы можно найти в приложении Ж.

1. Поток C++:
   а) представляет собой поток функционального управления;
   б) представляет собой поток данных из одного места в другое;
   в) ассоциирован с конкретным классом;
   г) представляет собой файл.
2. Базовым для большинства потоковых классов является класс ___________.
3. Назовите три потоковых класса, предназначенных для файлового ввода/вывода.
4. Напишите выражение, создающее объект salefile класса ofstream, и ассоциируйте его с файлом SALES.JUN.
5. Истинно ли утверждение о том, что некоторые потоки являются входными, а некоторые — выходными?
6. Напишите if, определяющий, достиг объект ifstream под названием foobar конца файла или же возникла ошибка.
7. Мы можем выводить текст в объект класса ofstream с использованием оператора вставки << потому, что:
   а) класс ofstream — это поток;
   б) оператор вставки работает с любыми классами;
   в) на самом деле вывод осуществляется в cout;
   г) оператор вставки перегружен в ofstream.
8. Напишите выражение, записывающее единичный символ в объект fileOut класса ofstream.
9. Для записи данных, содержащих переменные типа float, в объект типа ofstream необходимо использовать:
   а) оператор вставки;
   б) seekg();
   в) write();
   г) put().

10. Напишите выражение, считывающее содержимое объекта ifile класса ifstream, в массив buff.
11. Биты режимов, такие, как app и ate:
    а) определяются в классе ios;
    б) могут устанавливать, для чтения или для записи открыт файл;
    в) работают с функциями put() и get();
    г) устанавливают режимы открытия файлов.
12. Дайте определение термину *текущая позиция* в контексте работы с файлами.
13. Истинно ли утверждение о том, что файловый указатель всегда содержит адрес файла?
14. Напишите выражение, сдвигающее текущую позицию на 13 байтов назад в потоковом объекте f1.
15. Выражение f1.write((char*)&obj1, sizeof(obj1));:
    а) записывает методы obj1 в f1;
    б) записывает данные obj1 в f1;
    в) записывает методы и данные obj1 в f1;
    г) записывает адрес obj1 в f1.
16. Аргументы командной строки:
    а) это разборки в армии;
    б) набираются после названия программы в командной строке;
    в) делаются доступными с помощью аргументов main();
    г) доступны только дисковым файлам.
17. Что означает флаг skipws при его использовании с cin?
18. Напишите описатель для main(), позволяющий программе распознавать аргументы командной строки.
19. В консольных программах доступ к принтеру осуществляется с помощью зарезервированного имени ____________.
20. Напишите описатель перегруженного оператора >>, который берет данные из объекта класса istream и выводит их как данные объекта класса Sample.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Рассмотрите класс Distance из программы ENGLCON, глава 6 «Объекты и классы». Используя цикл, аналогичный приводимому в программе DISKFUN в этой главе, получите несколько значений от пользователя и запишите их в файл. Добавьте их к уже записанным там данным (при их наличии). При окончании пользователем ввода прочитайте файл и выведите на экран все значения.

*2. Напишите программу, эмулирующую команду COPY (MS DOS). То есть программа должна копировать содержимое одного файла в другой. Должно использоваться два аргумента командной строки — исходный файл и файл назначения. Например:

```
C:\>ocopy srcfile.cpp destfile.cpp
```

Осуществляйте проверку числа аргументов командной строки и возможность открытия указанных пользователем файлов.

*3. Напишите программу, возвращающую размер файла, указанного в командной строке:

```
C:\>filesize program.ext.
```

4. В цикле запрашивайте у пользователя данные, состоящие из имени, отчества, фамилии и номера работника (типа unsigned long). Затем осуществите форматированный вывод в объект ofstream с помощью оператора вставки (<<). Не забывайте, что строки данных должны оканчиваться пробелами или другими разделителями. Когда пользователь сообщит об окончании ввода, закройте объект ofstream, откройте объект ifstream, прочитайте и выведите на экран все данные из файла, после чего завершите программу.

5. Создайте класс time, включающий себя целые значения часов, минут и секунд. Напишите метод get_time(), спрашивающий время у пользователя, и метод put_time(), выводящий время в формате 12:59:59. Внесите в функцию get_time() проверку на ошибки, чтобы минимизировать возможность неправильного ввода пользователем. Эта функция должна отдельно спрашивать часы, минуты и секунды, проверяя каждое введенное значение на наличие флагов ошибок ios, а также проверяя, укладывается ли значение в заданный диапазон. Для часов диапазон составляет от 0 до 23, а для минут и секунд — от 0 до 59. Не вводите данные в виде символьных строк с последующим конвертированием. Вводите значения сразу же как int. Это означает, что вы не сможете выявлять записи с ненужными здесь десятичными запятыми, но это в данной программе не так важно.

   В main() используйте цикл для получения значений времени от пользователя функцией get_time(), затем для их вывода функцией put_time():

```
Введите часы: 8
Введите минуты: 2
Введите секунды: 39
Время = 8:02:39
Продолжить (y/n)? y
Введите часы: 28
Значение часов должно лежать между 0 и 23!
Введите часы: 1
Введите минуты: 10
Введите секунды: пять
Неправильно введены секунды!
Введите секунды: 5
Время = 1:10:05
```

6. Создайте класс name, включающий в себя данные из упражнения 4 (имя, отчество, фамилия и номер работника). Создайте методы для этого класса,

осуществляющие файловый ввод/вывод данных указанного класса (с использованием ofstream и ifstream). Используйте форматирование данных (операторы << и >>). Функции чтения и записи должны быть независимыми: в них необходимо внести выражения для открытия соответствующего потока, а также чтения и записи данных.

Функция записи может просто добавлять данные в конец файла. Функции чтения потребуется некоторое условие выборки конкретной записи. Можно вызывать ее с параметром, представляющим собой номер записи. Но как, даже зная, какую запись следует читать, функция найдет ее? Использование seekg() тут не поможет, так как при форматированном вводе/выводе все записи имеют разные размеры (в зависимости от количества символов в строке и разрядности числа). Поэтому придется просто считывать записи подряд, пока не будет найдена нужная.

В main() вставьте вызовы описанных выше методов, чтобы пользователь мог ввести данные с их последующей записью в файл. Затем программа должна выполнить чтение и продемонстрировать результаты этого чтения на экране.

7. Другим подходом к добавлению файлового потока к объекту является превращение самого этого потока в статическую компоненту объекта. Для чего это делается? Ну, часто бывает проще представить себе поток связанным с классом в целом, а не с отдельными его объектами. К тому же, гораздо правильнее открывать поток только один раз для записи и чтения всего, что нужно. Например, мы открываем файл и начинаем последовательное чтение. Функция чтения всякий раз возвращает данные для следующего объекта. Указатель файла сдвигается при этом автоматически, так как между чтениями файл мы не закрываем.

   Перепишите программы из упражнений 4 и 6 таким образом, чтобы использовать объект fstream в качестве статической компоненты класса name. Функционирование программы должно сохраниться. Напишите статическую функцию для открытия потока и еще одну — для сбрасывания файлового указателя и установки его на начало файла. Эту же функцию можно использовать для чтения всего файла.

8. Основываясь на программе LINKLIST из главы 10 «Указатели», напишите программу, позволяющую пользователю выбрать одно из четырех действий нажатием соответствующей кнопки. Действия таковы:

   - добавить ссылку в список (от пользователя требуется ввести целое число)
   - показать данные по всем ссылкам из списка;
   - записать в файл данные для всех ссылок (создание или переписывание файла);
   - считать все данные из файла и создать новый список ссылок, куда и поместить их.

   Первые два действия могут использовать методы, уже имеющиеся в LINKLIST. От вас требуется написать функции для чтения и записи файла. И для то-

го, и для другого можно использовать один и тот же файл. В нем должны храниться только данные; нет никакого смысла хранить содержимое указателей, которые, возможно, уже не будут нужны во время чтения списка.

9. Начните с упражнения 7 главы 8 «Перегрузка операций» и перегрузите операторы извлечения (>>) и вставки (<<) для класса frac в нашем калькуляторе с четырьмя действиями. Имейте в виду, что операторы могут связываться в цепочки, поэтому при выполнении действий с дробями понадобится только одно выражение: cin >> frac1 >> op >> frac2;

10. Добавьте к упражнению 9 проверку на наличие ошибок в операторе извлечения (>>). Но при этом, видимо, потребуется запрашивать сначала первую дробь, затем оператор, затем вторую дробь. Одним выражением, как в упражнении 9, уже будет не обойтись. Вывод сообщений об ошибках сделает работу с программой более понятной.

```
Введите первую дробь: 5/0
Знаменатель не может быть нулевым!
Введите первую дробь заново: 5/1
Введите оператор (+,-, *,/): +
Введите вторую дробь: одна треть
Ошибка ввода
Введите вторую дробь заново: 1/3
Ответ: -------------------------- 16/3
Продолжить (y/n)? n
```

Как показывает этот пример, необходимо следить за флагами ошибок ios и за тем, чтобы знаменатель не был равен нулю. Если возникает ошибка, пользователю должно быть предложено ввести данные еще раз.

11. Начните с класса bMoney, который мы последний раз видели в упражнении 5 главы 11. Перегрузите операторы извлечения и вставки, чтобы можно было осуществлять ввод/вывод объектов этого класса. Выполните какой-нибудь ввод/вывод в main().

12. К программе EMPL_IO из этой главы добавьте возможность поиска работника в списке, хранящемся в файле, по номеру. При нахождении совпадения нужно вывести данные об этом работнике на экран. Пользователь должен иметь возможность запустить функцию поиска нажатием клавиши 'f'. У пользователя спрашивается номер работника. Подумайте над вопросом, какой должна быть эта функция — статической, виртуальной или какой-то еще? Поиск и вывод на экран не должны пересекаться с данными в памяти.

ПРИМЕЧАНИЕ

Не пытайтесь прочитать файл, созданный программой EMPL_IO. Классы в программах разные благодаря методу find() в новой программе, и проблемы, которые могут возникнуть в случае, если их данные смешаются, уже обсуждались в этой главе. Вам может понадобиться включить параметр RTTI в компиляторе. Чтобы не возникало вопросов, следуйте указаниям, данным в приложении В «Microsoft Visual C++» и в приложении Г «Borland C++ Builder» (выбирайте то, что для вас актуальней).

Глава 13 Многофайловые программы

- Причины использования многофайловых программ
- Создание многофайловой программы
- Межфайловое взаимодействие
- Класс сверхбольших чисел
- Моделирование высотного лифта

В предыдущих главах мы видели, как связаны в C++ различные части программы — описатели классов, методы, функция main(). Но все программы, которые мы до сих пор рассматривали, состояли из одного файла. Теперь мы поднимемся в нашем искусстве программирования чуть выше и взглянем на многофайловые программы. Мы разберемся, как осуществляется связь между файлами и какая роль отводится заголовочным файлам.

Помимо теоретического обсуждения многофайловых программ, эта глава представит на суд читателя несколько сложных и больших программ. Вовсе не обязательно вникать во все мелочи, встречающиеся в этих программах; вы лишь должны хорошо усвоить принципы взаимодействия элементов сложных программ. В этих примерах показано, как классы могут использоваться в реальных приложениях: это уже не те маленькие программки, которые мы видели в предыдущих главах. С другой стороны, они не настолько длинные, чтобы потратить все лучшие годы жизни на их преодоление.

Причины использования многофайловых программ

Причин несколько. Важнейшими из них являются незаменимость библиотек классов, возможность использования принципа разделения работы над проектом на нескольких программистов, а также удобство создания концепции дизайна программ. Давайте рассмотрим эти вопросы.

Библиотеки классов

В традиционном процедурно-ориентированном программировании долгое время среди разработчиков ПО было принято предоставлять программистам библиотеки функций. С помощью комбинирования этих библиотек и добавления некоторых собственных процедур и функций получалась программа — продукт, представляемый конечному пользователю.

Библиотеки обычно содержат очень широкий спектр готовых функций, пригодных для различного применения. Например, разработчик может предложить библиотеку функций для статистической обработки данных или оптимизации работы с памятью компьютера.

Поскольку C++ построен на классах, а не на функциях, неудивительно, что библиотеки для программ на этом языке состоят из классов. Удивляет то, насколько библиотека классов лучше и прогрессивней старомодных библиотек функций. Поскольку классы представляют собой совокупность данных и функций для их обработки, а также из-за того, что они лучше моделируют реальную жизнь, интерфейс между библиотеками классов и приложениями, которые их используют, гораздо понятнее, чем между библиотеками функций и приложениями.

Поэтому библиотеки классов являются более важным предметом при программировании на C++, чем библиотеки функций при традиционном программировании. Использование этих библиотек освобождает программиста от очень многих забот. При разработке приложений оказывается, что если доступны необходимые библиотеки, то для создания полнофункционального продукта необходим минимум ручной работы по программированию. К тому же необходимо учитывать, что создается все больше и больше библиотек, а значит, все больший и больший спектр различного программного обеспечения можно создавать без лишних проблем.

Крайне важный пример библиотеки классов, который мы увидим в главе 15, это *Стандартная библиотека* C++ (*STL*).

Обычно библиотека классов состоит из интерфейса (interface) и реализации (implementation). Это две основные части, которые присутствуют практически всегда. Рассмотрим их подробнее.

Интерфейс

Назовем того человека, который написал библиотеку классов, *разработчиком библиотеки*, а того, кто использует эту библиотеку, — *программистом*.

Для того чтобы библиотеку можно было использовать, программисту необходим доступ к различным определениям, включая объявления классов. Они представляют собой общедоступную часть библиотеки и обычно поставляются в виде исходного кода в составе заголовочного файла с расширением .H. Обычно этот файл включается в текст программы с помощью директивы #include.

Объявления в заголовочных файлах должны быть общедоступными по нескольким причинам. Во-первых, программисту гораздо удобнее иметь перед глазами исходный текст, чем читать руководства и описания. Во-вторых (что

гораздо важнее), как иначе создавать объекты библиотечных классов, как вызывать библиотечные методы? Только с помощью объявлений классов в виде исходного кода это и возможно.

Эти определения называются интерфейсом, потому что это та часть библиотеки, которую программист видит и с помощью которой он взаимодействует с классами. При этом ему нет никакой нужды вникать в содержимое другой части библиотеки под названием реализация (implementation).

Реализация

Если интерфейсную часть можно назвать фасадом здания, то реализация представляет собой его внутренности. В контексте библиотеки классов реализация — это содержимое классов. Как туристу, любующемуся архитектурными памятниками, вовсе не обязательно знать, что находится внутри зданий, так и программисту не нужно вникать в то, как происходит внутренняя работа библиотечных классов. К тому же разработчики библиотек в большинстве случаев просто не предоставляют клиентам исходных кодов, так как они могут быть нелегально использованы или изменены. Поэтому методы классов чаще всего поставляются в виде объектных (.OBJ) или библиотечных (.LIB) файлов.

На рис. 13.1 показано, как соотносятся между собой компоненты многофайловых систем.

Организация и концептуализация

Программы могут разбиваться на части и из иных соображений, нежели работа с библиотеками классов. В общем случае проект разрабатывается группой программистов. Разграничив круг задач, поставленных перед каждым, можно решение каждой проблемы выделить в виде отдельного файла. Такой подход позволяет лучше организовывать работу команды и более четко определять взаимодействие частей программы.

К тому же, в соответствии с принципами технологии программирования, большие программы разбиваются на части в соответствии с функциональной направленностью. В одном файле может содержаться код поддержки графического вывода, в другом могут быть математические расчеты, а в третьем — дисковый ввод/вывод. При разработке больших программ держать все эти части в одном файле бывает затруднительно или просто невозможно.

Приемы работы с многофайловыми программами одинаковы, вне зависимости от причин разбиения.

Создание многофайловой программы

Допустим, у вас имеется библиотечный файл THEIRS.OBJ (работа с .LIB-файлами, впрочем, ничем не отличается от работы с .OBJ). К нему, соответственно, прилагается заголовочный файл THEIRS.H. В написанной вами программе MINE.CPP используются библиотечные классы. Как теперь прицепить одно к другому, соединить все три файла в одну исполняемую программу?

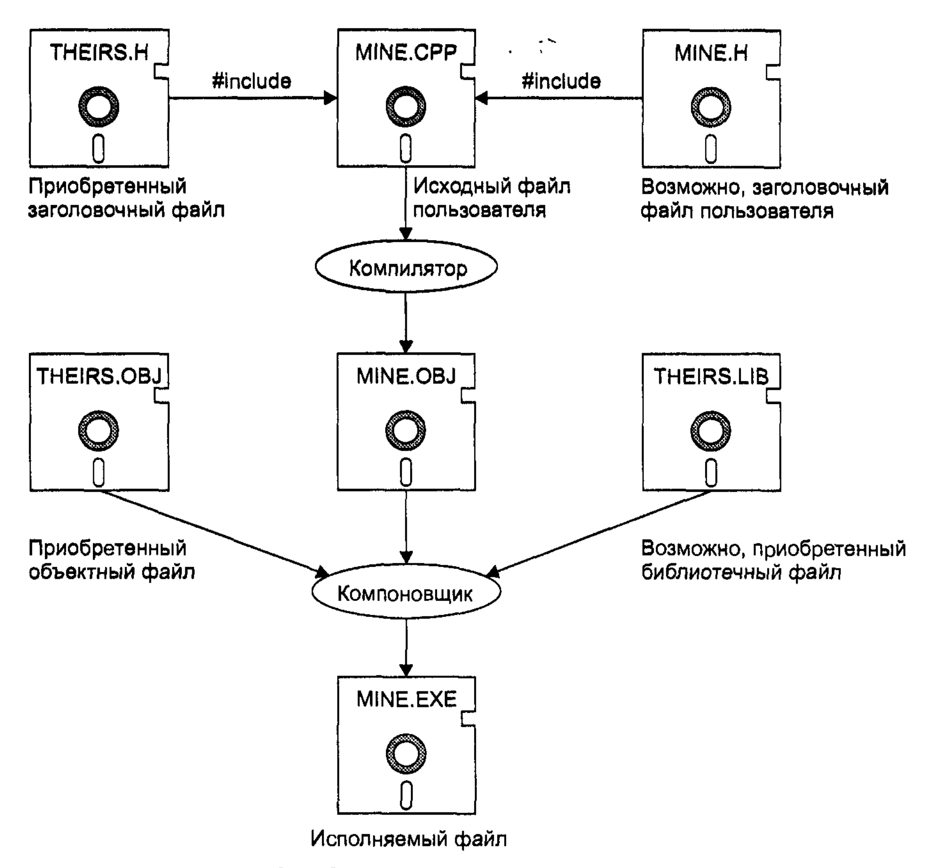


Рис. 13.1. Структура многофайловых приложений

Заголовочные файлы

Заголовочный файл THEIRS.H запросто вставляется в ваш исходный файл MINE.CPP с помощью директивы #include:

```
#include "theirs.h"
```

Кавычки вместо угловых скобок вокруг имени файла заставят компилятор искать файл в текущем каталоге, а не в каталоге включаемых файлов, указанном в настройках.

Директории

Убедитесь, что все компоненты — THEIRS.OBJ, THEIRS.H и MINE.CPP, — находятся в одном каталоге. Вообще говоря, лучше всего создавать для каждого проекта свой каталог, дабы избежать путаницы.

Каждый компилятор хранит свои собственные библиотечные файлы (такие, как IOSTREAM.H, CONIO.H) в определенном каталоге, часто называемом INCLUDE-

каталогом. Этот каталог может быть «закопан» довольно глубоко, но компилятор всегда знает, где его искать.

Кроме того, компилятору можно сказать о каталогах включаемых файлов, созданных программистом. Иногда хочется хранить включаемые файлы в каталогах, из которых они были бы доступны сразу нескольким проектам. В приложениях В и Г вы найдете информацию о том, как это делается.

Проекты

Большинство компиляторов управляют многофайловыми приложениями, называя их проектами. В проект входят все файлы, необходимые программе. В нем также содержатся инструкции по их компоновке, часто для этого заводится специальный файл, называемый *файлом проекта*. Расширение этого файла зависит от конкретной среды программирования и от ее производителя. В Borland-версии C++ это .BPR, а в Miscrosoft-версии — .DSP. Современные системы создают и обновляют этот файл автоматически, поэтому просто надо знать о его существовании. В принципе, можно сказать компилятору обо всех исходных файлах (.CPP), которые вы собираетесь включить в проект. Файлы .LIB и .OBJ также могут включаться в проект вручную. Все в тех же приложениях В и Г вы найдете детальную информацию по созданию файлов проектов для различных версий компиляторов.

Всего лишь одна команда требуется компилятору, чтобы откомпилировать все исходные и заголовочные файлы, скомпоновать .LIB и .OBJ файлы и создать исполняемый .EXE-файл. Это называется процессом *сборки*.

Одно из замечательных свойств проектов — они хранят данные о том, когда был откомпилирован каждый исходный файл. Перекомпилируются только те файлы, которые были изменены после последней сборки. Это определяется автоматически и позволяет ощутимо сэкономить время на компиляцию, что особенно заметно при сборке больших проектов. Некоторые компиляторы различают обычную сборку и сборку с перекомпиляцией измененных модулей. Обычная сборка перекомпилирует все файлы, независимо от даты внесения последних изменений.

Межфайловое взаимодействие

В многофайловых программах элементы, хранящиеся в разных файлах, должны каким-то образом сообщаться друг с другом. В этом разеделе мы выясним, как это делается. Вначале мы обсудим, как сообщаются откомпилированные по отдельности, но скомпонованные вместе исходные файлы (.CPP), а затем — как обращаться с заголовочными (.H) файлами.

Взаимодействие исходных файлов

Давайте рассмотрим, как сообщаются между собой отдельные исходные файлы. Мы обратим внимание на три основных элемента: переменные, функции и клас-

сы. Для каждого из них есть свои правила межфайлового взаимодействия. В этом месте полезно будет вспомнить о понятии зоны действия, для этого придется вернуться ненадолго к главе 5. Зона действия (или зона видимости) — это часть программы, внутри которой к данной переменной или другому элементу программы имеется доступ. Так, элементы, объявленные внутри функции, имеют *локальную* зону видимости, то есть доступ к ним может осуществляться только из данной функции. Соответственно, компоненты класса видны только внутри класса (кроме случаев использования оператора явного задания).

Элементы программы, объявленные вне всех функций и классов, имеют *глобальную* зону видимости, то есть доступ к ним осуществляется из любого места файла исходного текста программы. Как мы сейчас увидим, из других файлов того же проекта глобальные переменные тоже видны.

Межфайловые переменные

Начнем с простых переменных. Для этого вспомним разницу между объявлением и определением. Мы *объявляем*, декларируем какую-то простую переменную, задавая ее тип и имя. Это не означает, что для нее сразу же резервируется место в памяти. Объявление переменной просто сообщает компилятору о том, что где-либо там, в программе, может встретиться переменная с таким-то именем и такого-то типа. Переменная *определяется*, когда под нее в памяти резервируется место, которое способно содержать любое ее значение. Определение, так сказать, создает реальную переменную.

Большинство объявлений являются также и определениями. Точнее, единственным объявлением простой переменной, *не* являющимся определением, является объявление с использованием зарезервированного слова extern (без инициализации):

```
int SomeVar;            //объявление и определение в одном флаконе
extern int someVar;     //только объявление
```

Можно догадаться, что глобальная переменная может быть определена только один раз во всей программе, независимо от того, из скольких файлов она состоит.

```
//файл A
int globalVar;          //определение в файле A
//файл B
int globalVar;          //как НЕЛЬЗЯ делать: то же определение в
                        //файле B
```

Конечно, такие строгости касаются только глобальных переменных. Если переменные являются локальными по отношению к каким-либо классам или функциям, то вы можете сколько душе угодно определять одни и те же переменные с одинаковыми именами и типами данных. Не забудьте только разнести их по разным зонам видимости. Но лучше, конечно, вообще так не делайте — переменные с одинаковыми именами ничего, кроме сумятицы, в программу не вносят.

Как же обеспечить доступ к глобальным переменным, находящимся в одном файле, из другого? Тот факт, что компоновщик будет воротить нос при попытке

определения одной и той же глобальной переменной в разных файлах, еще не означает, что она будет отовсюду видна. Все-таки переменную нужно объявлять во всех файлах, в которых она используется. Например, такой вариант не подходит:

```
//файл A
int globalVar;    //Определение в файле A
//файл B
globalVar=3;      //НЕЛЬЗЯ! globalVar тут никто не знает
```

Компилятор справедливо заметит, что globalVar — неидентифицированный идентификатор.

Чтобы то, что мы так рекламировали в начале этого пункта, было действительно правдой, то есть чтобы переменная, определенная в одном файле, была видна в другом, нужно объявлять ее во всех остальных файлах с помощью зарезервированного слова extern.

```
//файл A
int globalVar;          //Определение в файле A
//файл B
extern int globalVar;   //Объявление в файле B
globalVar = 3;          //Вот теперь все хорошо
```

Как вы уже, наверное, догадались, объявление глобальной переменной в файле A сделало ее видимой в файле B. Зарезервированное слово extern означает, что объявление в данном случае — это только объявление, ничего более. Оно просит компилятор, который в каждый момент времени видит только один файл, не обращать внимание на то, что переменная globalVar не определена в файле B. Компоновщик, который с высоты своего особого статуса обозревает все файлы проекта, позаботится об установке ссылки на переменную, не определенную в данном файле.

Необходимо помнить об одном, возможно, неожиданном, ограничении: переменную нельзя инициализировать в объявлении с extern. Выражение

```
extern int globalVar = 27;   //не то, что вы имеете в виду
```

заставит компилятор думать, что вы хотите определить globalVar, а не просто объявить ее. То есть он просто проигнорирует слово extern и сделает из объявления определение. Если эта же переменная где-то в другом файле уже определена, то компоновщик выдаст ошибку повторного определения.

Но что, если вам действительно никак не прожить без глобальных переменных с одинаковыми именами в разных файлах? В этом случае необходимо определять их с помощью зарезервированного слова static. Тем самым область видимости глобальной переменной сужается до файла, в котором она определена. Другие переменные с тем же именем могут в других файлах использоваться без ограничений.

```
//файл A
static int globalVar;   //определение: переменная видна
                        //только в A
//файл B
static int globalVar;   //определение: переменная видна
                        //только в B
```

Хотя определены две переменные с одинаковыми именами, конфликта не возникает. Код, в который входит обращение к globalVar, будет обращаться только к переменной, определенной в данном файле. В этом случае говорят, что статическая переменная имеет *внутреннее связывание*. Нестатические глобальные переменные имеют *внешнее связывание*. (Как мы увидим чуть позднее, для сужения области видимости до данного файла может использоваться пространство имен.)

В многофайловых программах мы рекомендуем делать глобальные переменные статическими, если по логике работы к ним не требуется доступ более чем из одного файла. Это предотвратит возникновение ошибки, связанной со случайным заданием того же имени переменной в другом файле. К тому же листинг в этом случае становится более прозрачным — не нужно заботиться о проверке разных файлов на предмет совпадения имен.

Обратите внимание, что зарезервированное слово static имеет несколько значений в зависимости от контекста. В главе 5 «Функции» мы обсуждали, что когда static изменяет локальную переменную (то есть определенную внутри функции), изменяется ее продолжительность жизни от функции до всей программы, но видимость сохраняется неизменной, ограниченной функцией. Как уже говорилось в главе 6 «Объекты и классы», компонентные данные статических классов имеют одинаковое значение для всех объектов, а не для каждого — свое. Однако в контексте глобальных переменных слово static просто сужает область видимости до одного файла.

Переменная, определенная с помощью const, в общем случае не видна за пределами одного файла. В этом смысле она такая же, как static. Но ее можно сделать видимой из любого файла программы и для определения, и для объявления с помощью слова extern:

```
//файл A
extern const int conVar2 = 99:    //определение
//файл B
extern const int conVar2;         //объявление
```

Здесь файл B будет иметь доступ к переменной conVar2, определенной в файле A. Компилятор различает объявление и определение константы по наличию или отсутствию инициализации.

Межфайловые функции

Мы все помним, что объявление функции задает ее имя, тип возвращаемых данных и типы всех аргументов. Определение функции — это объявление плюс тело функции (тело функции — код, содержащийся внутри фигурных скобок).

Когда компилятор генерирует вызов функции, ему, в общем-то, незачем знать, как она работает. Все, что нужно знать, это ее имя, тип и типы ее аргументов. Все это есть в объявлении функции. Поэтому запросто можно определить функцию в одном файле, а создавать ее вызовы из другого. Никаких дополнительных разрешений (типа слова extern) не требуется. Нужно только объявить функцию в том файле, откуда будет производиться вызов.

```
//файл A
int add(int a, int b)   //определение функции
  {return a+b; }        // (с телом функции)
```

```
//файл B
int add(int,int);       //объявление функции (без тела)
. . .
int answer = add(2, 3); //вызов функции
```

Зарезервированное слово extern с функциями не используется, так как компилятор в состоянии отличить определение от объявления по наличию или отсутствию тела функции.

Можно совершенно случайно объявить (*не определить*!) функцию или другой элемент программы несколько раз. Компилятор сохраняет спокойствие, но только до тех пор, пока объявления не противоречат друг другу.

```
//файл A
int add(int,int);       //объявление функции (без тела)
int add(int,int);       //второе объявление. Пусть будет.
```

Подобно переменным, функции могут быть сделаны невидимыми для других файлов. Для этого при их объявлении используется то же самое слово static.

```
//файл A
static int add(int a, int b); //определение функции
{return a+b; }

//файл B
static int add(int a, int b); //другая функция
{return a+b; }
```

Этот код создает две разные функции. Ни одна из них не видна вне того файла, в котором она определена.

Межфайловые классы

Классы отличаются от простых переменных тем, что определение класса не подразумевает резервирования памяти. Оно разве что информирует компилятор о том, что именно входит в класс. Это примерно как оговаривать, сколько байтов резервировать под переменную типа int, с той лишь разницей, что компилятор уже знаком с int, но не знаком с типом someClass, пока вы его не определите.

Определение класса содержит определения или объявления всех его членов:

```
class someClass            //определение класса
  {
  private:
    int memVar;            //определение компонентной переменной
  public:
    int memFunc(int, int); //объявление метода
  };
```

Компоненты класса должны быть объявлены, но не обязательно определены. Как известно, определения методов помещаются вне класса и идентифицируются с помощью оператора разрешения контекста.

Объявление класса лишь говорит о том, что то или иное имя принадлежит данному классу. В нем ничего не сообщается компилятору о компонентах класса.

```
class someClass;      //объявление класса
```

Не путайте определение класса с определением (созданием) объекта класса:

```
someClass anObj:
```

В отличие от определения класса, определение объекта подразумевает резервирование памяти для его размещения.

Классы ведут себя в межфайловых отношениях не так, как переменные и функции. Чтобы иметь доступ к классу из любого файла, входящего в программу, необходимо *определять* (а не просто объявлять) класс в каждом файле. Определение класса в файле А и объявление его в файле В не означает, что компилятор сможет создать в файле В объекты данного класса.

Почему же все так строго с классами? Дело в том, что компилятору необходимо знать тип данных всего, что он компилирует. Объявления для переменных достаточно потому, что в нем указывается уже известный тип.

```
//объявление
extern int someVar:       //видя объявление.
someVar = 3:              // компилятор может обработать это.
```

Объявление функции тоже рассказывает компилятору обо всех типах используемых данных.

```
//объявление
int someFunc(int. int):     //видя объявление.
var1 = someFunc(var2.var3): //компилятор может обработать это.
```

Что касается класса, то требуется именно его определение, чтобы указать все типы используемых данных и методов.

```
//определение
class someClass:                //видя определение. компилятор
  {
  private:
    int memVar:
  public:
    int memFunc(int. int):
  }:
someClass someObj:              //может обработать это
v1 = someObj.memFunc(v2. v3):   //и это
```

Одного объявления, как видите, недостаточно компилятору для генерации кода, который мог бы работать с объектами класса (за исключением указателей и ссылок на объекты).

Невозможно определить класс дважды в одном исходном файле, но каждый исходный файл может иметь свое определение того же класса. На самом деле, конечно, файлу нужно определение класса, только если в нем этот класс используется. В следующем параграфе мы покажем более правильный способ межфайловой коммуникации классов — с использованием заголовочных файлов.

Заголовочные файлы

Как отмечалось в главе 2, директива #include работает, как функция ВСТАВИТЬ в текстовых редакторах, то есть указанный после этой директивы файл просто

вставляется в исходный. В очень многих примерах мы видели включения библиотечных файлов типа IOSTREAM.

Можно также и самостоятельно написать заголовочный файл и включить его в свою программу.

Общая информация

Одной из причин использования заголовочных файлов является возможность вставки одной и той же информации в разные файлы. В них содержатся объявления переменных или функций, которые и включаются в исходные тексты программ. Тем самым можно открыть доступ к этим переменным и функциям из множества файлов.

Конечно, каждый элемент программы должен быть определен в каком-то месте. Например, переменная и функция объявлены в fileH.h, а определены в fileA.cpp. Код из файла fileB.cpp может использовать эти элементы без дополнительных объявлений.

```
//fileH.h
extern int gloVar;              //объявление переменной
int gloFunc(int);               //объявление функции

//fileA.cpp
int gloVar;                     //определение переменной
int gloFunc(int n)              //определение функции
  { return n; }

//fileB.cpp
#include "fileH.h"
. . .
gloVar = 5;                    //работа с переменной
int gloVraB=gloFunc(gloVar);  //работа с функцией
```

Помните, что в заголовочном файле можно хранить объявления, но не определения переменных или функций. При использовании этого файла в других (если только данные не static или const) такая оплошность приведет к ошибке компоновщика «повторные определения».

Одним из основных методов является следующий. В заголовочный файл вносится определение класса, который используется при необходимости всеми исходными файлами. Это не приводит к ошибке повторного определения, потому что определение класса не означает резервирования памяти — это только спецификация его компонентов.

```
//fileH.h
class someClass;                 //определение класса
  {
  private:
    int memVar;
  public:
    int memFunc(int, int);
  };
//fileA.cpp
#include "fileH.h"
int main()
  {
```

```
someClass Obj1;              //создание обьекта
int var1 = Obj1.memFunc(2, 3); //работа с обьектом
   }
//fileB.cpp
#include "fileH.h"
int func()
  {
someClass Obj2;              //создание обьекта
int var2 = Obj2.memFunc(4, 5); //работа с обьектом
   }
```

А что бы, интересно, было, если бы вместо использования заголовочного файла мы просто вставили этот текст с определением класса в каждый из файлов? Ничего бы страшного, на самом деле, не произошло, просто каждое маленькое изменение в классе пришлось бы производить во всех этих файлах. Это отняло бы массу времени, да еще и наверняка привело бы к опечаткам.

До сих пор мы демонстрировали определения классов без внешних определений их методов. Но куда же нам вставить, собственно, основное содержимое классов — определения методов? Как и для любых других функций, это может быть сделано в любом исходном файле, и компоновщик соединит все вместе как нужно. На то он и компоновщик. Собственно говоря, определение класса служит для того, чтобы можно было объявлять методы в каждом файле. Как и в программах, состоящих из единственного файла, определения методов должны включать в себя имя класса и оператор разрешения контекста. Пример:

```
//fileH.h
class someClass;            //определение класса
  {
  private:
    int memVar;
  public:
    int memFunc(int, int);  //объявление метода
  };

//fileA.cpp
#include "fileH.h"
int someClass::memFunc(int n1, int n2); //определение метода
   { return n1+n2; }

//fileB.cpp
#include "fileH.h"
someClass anObj;                    //создание обьекта
int answer = anObj.memFunc(6, 7);   //работа метода
```

Ошибка повторения включений

Мы упоминали, что нельзя определять функцию или переменную в заголовочном файле, который будет использован несколькими исходными файлами. Это приводит к ошибкам повторных определений. Подобная проблема возникает и тогда, когда по ошибке включают один и тот же заголовочный файл дважды. Как такое может случиться? Да запросто:

```
//файл app.cpp
#include "headone.h"
#include "headone.h"
```

Но это еще ничего. Представьте себе, что у вас есть исходный файл app.cpp и два заголовочных — headone.h и headtwo.h. К тому же headone.h включает в себя headtwo.h! К сожалению, про это обстоятельство забывают и включают оба файла в app.cpp:

```
//файл headtwo.h
int globalVar;

//файл headone.h
#include "headtwo.h"

//файл app.cpp
#include "headone.h"
#include "headtwo.h"
```

Что теперь будет после компиляции app.cpp? Так как директивой #include мы вставили заголовочные файлы, реальное содержимое исходного файла будет таково:

```
//файл app.cpp
. . .
int globalVar;   //из headtwo.h через headone.h
. . .
int globalVar;   //напрямую из headtwo.h
```

Разумеется, компилятор сообщит, что globalVar определена дважды.

Предупреждение ошибок повторения включений

С рассеянностью приходится бороться. Вот как можно предупредить ошибки повторных определений даже при ошибках повторения включений: определения в заголовочном файле следует начинать с директивы препроцессора:

```
#if !defined( HEADCOM )
```

(На месте HEADCOM может быть любой идентификатор.) Это выражение говорит о том, что если HEADCOM еще не был определен (восклицательный знак означает логическое отрицание), то весь текст отсюда и до обеда, точнее, до #endif (закрывающая директива для #if), будет просто вставляться в исходный файл. В противном случае (если HEADCOM уже определен ранее, в чем можно удостовериться с помощью директивы #define HEADCOM) следующий за #if текст не будет включен в исходный код. Как говорят в американских фильмах, он погиб в этой мясорубке. Поскольку переменная HEADCOM не была определена до того, как эта директива встретилась впервые, но сразу же после #if !defined() оказалась определенной, весь текст, заключенный между #if и #endif, будет включен один раз, но это будет первый и последний раз. Вот как это делается:

```
#if !defined( HEADCOM ) //Если HEADCOM еще не определен,
#define HEADCOM         //определить ее

int globalVar;          //определить переменную
int func(int a, int b)  //определить функцию
  { return a+b; }
#endif                  //закрывающая условие директива
```

Этот подход следует использовать всегда, когда существует возможность случайно включить заголовочный файл в исходный более одного раза.

Раньше использовалась директива #ifndef, это то же самое, что #if !defined(); вы можете встретить ее во многих заголовочных файлах, входящих в комплект поставки вашего компилятора. Тем не менее от ее использования теперь отказались.

Понятно, что эта «защита от дурака» с использованием #if !defined() сработает только в том случае, если определение globalVar (или любой другой переменной или функции) может случайно быть включено несколько раз в один и тот же исходный файл. Она не сработает, если globalVar определена в .H-файле, и его включают в разные файлы A и B. Препроцессор бессилен в этом случае, он не может определить наличие одинаковых выражений в отдельных файлах, поэтому все станет известно компоновщику, который, конечно же, нажалуется на то, что globalVar определена несколько раз.

Пространства имен

Мы уже знаем, как ограничить видимость элементов программ, объявляя их внутри файла или класса либо делая их static или const. Иногда, тем не менее, требуется более гибкий подход к этому вопросу.

Например, при написании библиотеки классов программисты предпочитают использовать короткие и понятные имена для функций, не являющихся методами, и отдельных классов. Например, add(), book. А эти имена уже могут быть выбраны другими разработчиками классов в их библиотеках или программистами, использующими в своих программах данную библиотеку. Это может привести к «столкновению имен», и вы долго будете ломать голову над тем, откуда берется сообщение компилятора о повторном определении. Перед введением понятия пространства имен программистам приходилось использовать длинные имена во избежание этой коллизии:

```
Sancho's_Simplified_Statistics_Library_Forever_add();
```

Не нужно объяснять, что использование подобных имен переменных, функций, классов сильно усложняет восприятие листингов, усложняет работу программистов, а сами листинги неоправданно раздуваются в объеме. Пространства имен решают эту проблему (надо отметить, что обычные методы классов не могут вызывать столкновения имен, так как их видимость ограничена классом).

Определение пространства имен

Пространство имен — это некая именованная область файла. Следующий код определяет пространство имен geo с некоторыми объявлениями внутри него.

```
namespace geo
  {
  const double PI = 3.14159
double circumf(double radius)
  { return 2 * PI * radius; }
  } //конец пространства geo
```

Фигурные скобки ограничивают пространство имен. Переменные и другие элементы программы, объявленные между ними, называются членами пространства имен. Обратите внимание: за закрывающей фигурной скобкой не следует точка с запятой, как в классах.

Доступ к членам пространств имен

Часть кода, находящаяся вне пространства имен, не имеет доступа к его членам, по крайней мере, обычным способом. Пространство имен делает их невидимыми:

```
namespace geo
  {
  const double PI = 3.14159
double circumf(double radius)
  { return 2 * PI * radius; }
  } //конец пространства geo
double c = circumf(10);       //здесь это не сработает
```

Чтобы иметь доступ к элементам пространства имен извне, необходимо при обращении к ним использовать название этого пространства. Это можно сделать двумя способами. Во-первых, каждый элемент можно предварять названием пространства и оператором разрешения контекста:

```
double c = geo::circumf(10); //полный порядок
```

Во-вторых, можно использовать директиву using:

```
using namespace geo;
double c = circumf(10);       //полный порядок
```

Директива using обычно делает пространство имен видимым начиная с места ее написания и до конца. Можно сузить область действия директивы до блока, используя ее, например, в функции:

```
void seriousCalcs()
  {
  using namespace geo;
  //идет какой-то код
  double c = circumf(r);     //OK
  }
double c = circumf(r);       //Не работает
```

Как видим, здесь члены пространства имен видны только внутри тела функции.

Пространства имен в заголовочных файлах

Пространства имен часто используются в заголовочных файлах библиотек классов или функций. Каждая библиотека может иметь свое пространство имен. На данный момент вы уже знакомы с пространством std, чьи члены составляют Стандартную библиотеку C++.

Неоднократное определение пространств имен

Определения пространств имен могут встречаться в тексте программы несколько раз:

```
namespace geo
  {
  const double PI = 3.14159
  } //конец пространства geo
//тут какой-то код
namespace geo
  {
double circumf(double radius)
  { return 2 * PI * radius; }
  } //конец пространства geo
```

Это выглядит как двойное определение пространства, но на самом деле это просто продолжение того же определения. Эта маленькая хитрость позволяет использовать одно и то же пространство имен в нескольких заголовочных файлах. В Стандартной библиотеке C++ примерно дюжина заголовочных файлов использует пространство std.

```
//fileA.h
namespace alpha
  {
  void funcA();
  }

//fileB.h
namespcae alpha
  {
  void funcB();
  }

//файл Main.cpp
#include "fileA.h"
#include "fileB.h"
using namespace alpha;
funcA();
funcB();
```

Объявления можно помещать и вне пространства имен, и они будут работать, как если бы они были внутри него. Все, что для этого нужно, это имя пространства и оператор разрешения контекста:

```
namespace beta
  {
  int uno
  }
int beta::dos;
```

В этом примере и uno, и dos являются объявлениями в пространстве имен **beta**.

Неименованные пространства имен

Можно создать пространство имен и не присваивая ему имени. При этом автоматически создастся пространство, элементы которого видны только из данного файла. Компилятор дает ему имя, совпадающее с именем файла. Члены неименованного пространства имен видны отовсюду внутри файла. В следующем лис-

тинге функции funcA() и funcB() имеют доступ к переменной gloVar, каждая из своего файла.

```
//fileA.cpp
namespace            //неименованное пространство. для fileA.cpp
  {
  int gloVar = 111;
  }
funcA()
  {cout << gloVar; } //Выводит: 111

//fileB.cpp
namespace            //неименованное пространство. для fileB.cpp
  {
  int gloVar = 222;
  }
funcB()
  {cout << gloVar; } //Выводит: 222
```

В данном примере оба файла содержат переменную gloVar, но конфликта имен не возникает, так как переменные объявлены в неименованных пространствах имен и находятся в разных файлах.

Это как бы альтернатива использованию static для уменьшения зоны видимости глобальных переменных до размеров одного файла. На самом деле использование пространств имен сейчас признается более предпочтительным, нежели использование static-элементов.

Переименование типов с помощью typedef

Зарезервированное слово typedef может оказаться полезным в определенных ситуациях, и вы наверняка встречали его, по крайней мере, в чужих листингах. С его помощью можно создать новое имя для типа данных. Например, выражение

```
typedef unsigned long unlong;
```

переименовывает unsigned long в unlong, точнее, делает их синонимами. Теперь можно объявлять переменные с использованием нового имени:

```
unlong var1, var2;
```

Это может помочь сэкономить немножко места. С большей пользой можно применить это переименование для того, чтобы расширить назначение переменных этого типа:

```
typedef int FLAG;       //переменные типа int для хранения флагов
typedef int KILOGRAMS;  //переменные типа int для килограммов
```

Если вам не нравится, как в C++ задаются указатели, можно изменить их объявление:

```
int *p1, *p2, *p3;      //обычное объявление
typedef int* ptrInt;    //новое имя для указателя на int
ptrInt p1, p2, p3;      //упрощенное объявление
```

Вот так. Чтобы не писать эти надоедливые звездочки.

Так как классы являются типами в C++, можно использовать typedef для их переименования. Ранее мы говорили, что иногда приходится называть данные длинными именами. Писать их всякий раз — утомительно и долго. Но, не переименовывая сам класс, можно создать синоним его имени, так сказать, задать уменьшительно-ласкательное его название с помощью typedef:

```
class GeorgeSmith_Display_Utility        //определение класса
{
//члены класса
};

typedef GeorgeSmith_Display_Utility GSdu: //новое имя класса
GSdu anObj:              //создать объект, используя новое имя
```

Переименование обычно используется в заголовочных файлах для того, чтобы новые имена могли использоваться множеством исходных файлов. Многие разработчики ПО настолько любят typedef, что в результате их переименований получается практически другой язык программирования.

Теперь, после изучения некоторых теоретических положений, касающихся многофайловых программ, можно перейти к практическим примерам. В них, конечно, не охвачены все темы, которые мы рассматривали в этой главе, но они показывают общую концепцию работы программиста с программами, состоящими из нескольких файлов.

Класс сверхбольших чисел

Иногда даже типа unsigned long не хватает для точного выполнения некоторых арифметических операций, хотя этот класс имеет наибольшую разрядность среди всех остальных. В нем могут содержаться числа величиной до 4 294 967 295, это 10 десятичных разрядов, то есть примерно столько же, сколько в карманном калькуляторе. Но если приходится работать с числами, содержащими большее число значащих разрядов, то возникает проблема.

В следующем примере мы создадим класс, который может содержать до 1000 десятичных разрядов! Впрочем, если требуется хранить большее число, надо просто поменять одну константу в программе.

Числа как строки

Класс verylong хранит числа в виде строк разрядов. Это строки старомодного типа char* из языка C. В этом контексте с ними проще работать, чем с типом string. Именно использованием этого уже подзабытого типа объясняется такая большая разрядность: C++ может рассматривать эти строки просто как массивы. Таким образом, можно хранить в строках числа сколь угодно большой разрядности. В классе verylong есть два компонентных данных: массив типа char, хранящий строку разрядов, и целочисленное значение длины строки (последнее, строго говоря, не столь необходимо, но использовать его как-то проще, чем постоянно

вычислять strlen() для нахождения длины строки). Цифры (собственно, разряды) в строке хранятся в обратном порядке, таким образом, последняя значащая цифра содержится в vlstr[0]. Это упрощает выполнение операций над строкой. На рис. 13.2 показано, как число хранится в строке.

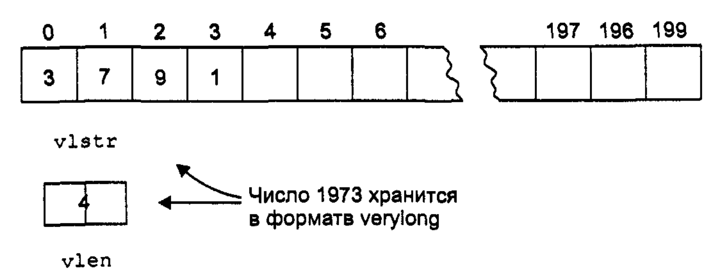


Рис. 13.2. Число типа verylong

Мы включили в программу возможность складывать и умножать числа в формате verylong. В качестве упражнения можем предложить добавить функции вычитания и деления.

Описатель класса

Прежде всего рассмотрим заголовочный файл для VERYLONG. В нем содержится спецификация одноименного класса.

Листинг 13.1. Заголовочный файл для VERYLONG[1]

```
// verylong.h
// описатель класса сверхбольших целых чисел
#include <iostream>
#include <string.h>          //для strlen()и т. п.
#include <stdlib.h>          //для ltoa()
using namespace std:

const int SZ = 1000:         //максимальное число разрядов

class verylong
  {
  private:
    char vlstr[SZ]:          //число как строка
    int vlen:                //длина строки verylong
    verylong multdigit(const int) const: //прототипы
    verylong mult10(const verylong) const:// скрытых
                                         // функций
  public:
    verylong() : vlen(0)     //конструктор без аргументов
      { vlstr[0]='\0': }
    verylong(const char s[SZ]) //конструктор (1 аргумент)
      { strcpy(vlstr, s): vlen=strlen(s): } //для строки
    verylong(const unsigned long n) //конструктор (1 арг.)
      {                          //для long int
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
    ltoa(n, vlstr, 10);          //перевести в строку
    strrev(vlstr);               //перевернуть ее
    vlen=strlen(vlstr);          //найти длину
    }
  void putvl() const;            //вывести число
  void getvl();       //получить число от пользователя
  verylong operator + (const verylong); //сложить числа
  verylong operator * (const verylong); //умножить
};
```

В дополнение к данным в классе verylong имеются две скрытые функции. Одна из них умножает сверхбольшое число на цифру, другая — умножает на 10. Это внутренние функции, используемые при умножении сверхбольших чисел.

Можно заметить, что в классе три конструктора. Один сбрасывает сверхбольшое число в ноль, устанавливая в начале массива нуль-ограничитель и присваивая длине строки нулевое значение. Второй инициализирует число строкой (в которой оно хранится в обратном порядке), а третий — значением long int.

Метод putvl() выводит на экран значение числа, а getvl() — получает его от пользователя. Ввести можно сколь угодно большое число в пределах 1000 разрядов. Имейте в виду, что в этой программе нет никаких проверок на ошибки. Если вы напишете вместо цифр какие-то символы, программа не отреагирует на это, но и результат операции будет неправильный.

Два перегруженных оператора — «+» и «*» — выполняют, соответственно, сложение и умножение. Можно использовать даже такие сложные выражения, как

```
alpha = beta * gamma + delta
```

для выполнения арифметических действий над данными типа verylong.

Методы

Ниже приводится файл VERYLONG.CPP, содержащий определения методов.

Листинг 13.2. Реализационная часть класса verylong

```
// verylong.cpp
// реализация обработки данных типа verylong
#include "verylong.h"       //заголовочный файл для verylong
//-----------------------------------------------------------
void verylong::putvl() const //вывод на экран verylong
  {
  char temp[SZ];
  strcpy(temp,vlstr);        //создать копию
  cout << strrev(temp);      //перевернуть копию
  }                          //и вывести ее
//-----------------------------------------------------------
void verylong::getvl()       //получить сверхбольшое число от
                             //пользователя
  {
  cin >> vlstr;              //получить строку от пользователя
  vlen = strlen(vlstr);      //найти ее длину
  strrev(vlstr);             //перевернуть ее
  }
```

продолжение ⇗

Листинг 13.2 (*продолжение*)

```
//---------------------------------------------------------
verylong verylong::operator + (const verylong v)
                              //сложение
  {
  char temp[SZ];
  int j;
                              //найти самое длинное число
  int maxlen = (vlen > v.vlen) ? vlen : v.vlen;
  int carry = 0;              //установить в 1, если сумма >= 10
  for(j = 0; j<maxlen; j++)   //и так для каждой позиции
    {
    int d1 = (j > vlen-1)  ? 0 : vlstr[j]-'0';    //получить
                                                  //разряд
    int d2 = (j > v.vlen-1) ? 0 : v.vlstr[j]-'0'; //и еще
    int digitsum = d1 + d2 + carry;  //сложить разряды
    if( digitsum >= 10 )             //если перенос, то
      { digitsum -= 10; carry=1; }   //увеличить сумму на 10
    else                      //установить перенос в 1
      carry = 0;              //иначе перенос = 0
    temp[j] = digitsum+'0';   //вставить символ в строку
    }
  if(carry==1)                //если перенос в конце,
    temp[j++] = '1';          //последняя цифра = 1
  temp[j] = '\0';             //поставить ограничитель строки
  return verylong(temp);      //вернуть временный verylong
  }
//---------------------------------------------------------
verylong verylong::operator * (const verylong v)//умножение
  {                                             //сверхбольших чисел
  verylong pprod;             //произведение одного разряда
  verylong tempsum;           //текущая сумма
  for(int j=0; j<v.vlen; j++)//для каждого разряда аргумента
    {
    int digit = v.vlstr[j]-'0'; //получить разряд
    pprod = multdigit(digit);   //умножить текущий на него
    for(int k=0; k<j; k++)      //умножить результат на
      pprod = mult10(pprod);    //  степень 10-ти
    tempsum = tempsum + pprod;  //прибавить произведение к
                                //текущей сумме
    }
  return tempsum;               //вернуть полученную текущую сумму
  }
//---------------------------------------------------------
verylong verylong::mult10(const verylong v) const //умножение аргумента
                                                  //на 10
  {
  char temp[SZ];
  for(int j=v.vlen-1; j>=0; j--)//сдвинуться на один разряд
    temp[j+1] = v.vlstr[j];     //выше
  temp[0] = '0';                //обнулить самый младший разряд
  temp[v.vlen+1] = '\0';        //поставить ограничитель строки
  return verylong(temp);        //вернуть результат
  }
```

```
//--------------------------------------------------------
verylong verylong::multdigit(const int d2) const
  {                            //умножение числа на
  char temp[SZ]:               //аргумент (цифру)
  int j. carry = 0:
  for(j = 0: j<vlen: j++)      //для каждого разряда
    {             //  в этом сверхбольшом
    int d1 = vlstr[j]-'0':     //получить значение разряда
    int digitprod = d1 * d2:   //умножить на цифру
    digitprod += carry:        //добавить старый перенос
    if( digitprod >= 10 )      //если возник новый перенос,
      {
      carry = digitprod/10:    //переносу присвоить //значение старшего разряда
      digitprod -= carry*10:   //результату - младшего
      }
    else
      carry = 0:               //иначе перенос = 0
    temp[j] = digitprod+'0':   //вставить символ в строку
    }
  if(carry != 0)               //если на конце перенос,
    temp[j++] = carry+'0':     //это последний разряд
  temp[j] = '\0':              //поставить ограничитель
  return verylong(temp):       //вернуть сверхбольшое число
  }
```

Функции putvl() и getvl() устроены довольно просто. Они используют библиотечную функцию С под названием strrev() для переворачивания строки, но вводимые числа отображаются в нормальном порядке.

Перегруженная функция operator+() складывает два числа в формате verylong, а результат сохраняет в третьем числе. Сложение производится поразрядно. Оно начинается со сложения нулевого разряда обоих чисел, при необходимости запоминается перенос. Затем складываются первые разряды, добавляется перенос (если таковой имел место), при необходимости запоминается новый перенос. Операция продолжается до тех пор, пока не будут сложены все разряды в большем из двух чисел. Если числа разной длины, недостающие разряды из меньшего числа устанавливаются в ноль перед их сложением. На рис. 13.3 показан процесс сложения.

Соответственно, для умножения используется функция operator*(). Она умножает множимое (придется вспомнить давние годы, первые классы школы: множимое — это число, записываемое сверху при умножении «в столбик») на каждый разряд множителя (множитель — это то, что пишут снизу). Для этого вызывается функция multdigit(). Затем результат умножается на 10 нужное число раз (для сдвига окончательного результата на соответствующее число разрядов) с помощью функции mult10(). Итоги этих двух отдельных вычислений складываются с использованием описанной выше функции operator+().

Прикладная программа

Для тестирования всего, что было написано выше, можно использовать, например, программу FACTOR из главы 3 «Циклы и ветвления». Там мы вычисляли факториал введенного пользователем числа.

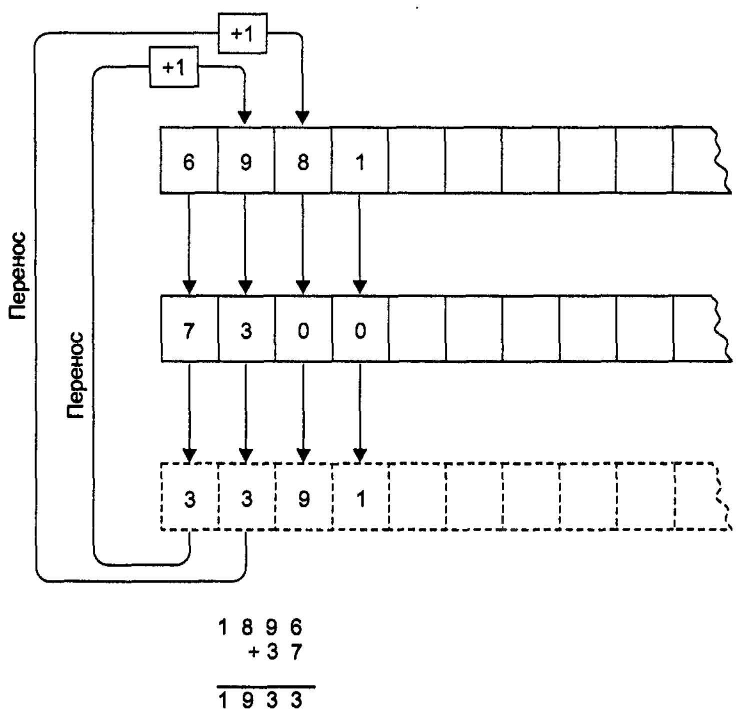


Рис. 13.3. Сложение чисел в формате verylong

Листинг 13.3. Программа VL_APP

```
// vl_app.cpp
// вычисляет факториалы больших чисел
#include "verylong.h"    //заголовочный файл verylong

int main()
  {
  unsigned long numb, j;
  verylong fact=1;        //инициализировать verylong

  cout << "\n\nВведите число: ";
  cin >> numb;            //ввод числа типа long int

  for(j=numb; j>0; j--)  //факториал - это numb *
    fact = fact * j;     //  numb-1 * numb-2 *
  cout << "Факториал= "; //  numb-3 и т. д.
  fact.putvl();           //вывести значение факториала
  cout << endl;
  return 0;
  }
```

В этой программе fact — переменная класса verylong. Другие переменные — numb, j — не обязаны быть того же типа, они еще не доросли до таких размеров. Например, для подсчета факториала числа 100 необходимо всего 3 разряда, тогда как в fact для этого должно быть 158 разрядов.

Обратите внимание, как в выражении

```
fact = fact * j;
```

переменная j автоматически переводится в формат verylong. Это делается с помощью конструктора с одним аргументом перед началом выполнения операции умножения.

Вот какую прелесть вы увидите на экране при нахождении факториала числа 100:

```
Введите число: 100
Факториал=
93326215443944152681699238856266700490715948264381621468592963895217599993229915608941463976
156518286253697920827223758251185210916864000000000000000000000000
```

А попробуйте-ка .вычислить *такое*, используя обычный тип long! Удивительно, но программа работает довольно быстро. Приведенный выше пример вычислялся какие-то доли секунды. Вы можете вычислять факториалы вплоть до 400!, примерно после этого числа будет достигнут 1000-разрядный предел формата[1].

Моделирование высотного лифта

Когда вы станете великим программистом и будете работать где-нибудь в Силиконовой долине, стоя в ожидании лифта внутри огромного офисного небоскреба, вспомните этот пример. Вы задумывались когда-нибудь о том, как лифт узнает, на какой этаж ему ехать? В старые добрые времена, конечно же, для этой цели существовали импозантные лифтеры (помните: «Доброе утро, Андрей Владимирович!», «Доброе утро, Юра!»), которым пассажиры сообщали нужный номер этажа. На панели внутри лифта при его движении загорались лампочки с номерами проезжаемых в данный момент этажей. Лифтеры выбирали, куда ехать, в соответствии с просьбами пассажиров и показаниями лампочек.

В наши дни прогресс уже дошел до того, что лифты сами соображают, куда им ехать и где останавливаться. В нашем следующем примере с помощью классов C++ моделируется система лифтов.

Из чего должна состоять такая система? Обычно в большом здании располагается несколько одинаковых лифтов. На каждом этаже есть кнопки «вниз» и «вверх». Обратите внимание, что в большинстве случаев на этаж выделяется одна пара таких кнопок. Когда вы вызываете лифт, вы не знаете, какой именно из них приедет раньше. Внутри лифта кнопочек существенно больше — одна для каждого этажа. Войдя в лифт, пассажиры обычно нажимают кнопку нужного этажа. Программа моделирует все эти составляющие процесса.

---

[1] Улучшить программу за счет сокращения затрат памяти на хранение таких огромных чисел можно, если использовать не десятичную, а более выгодные системы счисления — шестнадцатеричную, двухсотпятидесятишестиричную и т. п. (предел основания системы счисления зависит только от выбранной таблицы используемых символов). Однако при этом возникают свои сложности. — *Примеч. перев.*

Работа программы ELEV

При запуске вы увидите четыре лифта, находящихся внизу экрана, а также вертикальный список чисел слева от 1 до 20, увеличивающихся снизу вверх. Изначально все лифты на первом этаже. Это все показано на рис. 13.4

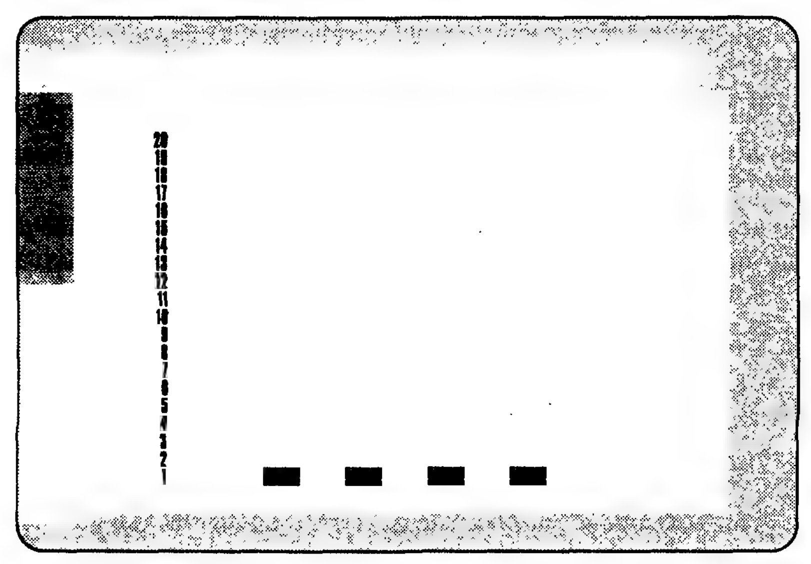


Рис. 13.4. Начальное состояние программы ELEV

Запрос этажа

При нажатии Enter внизу экрана появится текст:

```
На каком Вы этаже?
```

Можно ввести любое число от 1 до 20. Если вы только что приехали на работу и намерены подняться к своему рабочему месту, нажимайте 1. Если спешите спуститься на обед со своего этажа, введите его номер. Следующее, о чем спросит программа, это

```
В каком направлении будете двигаться (u/d):
```

Даем подсказку: если вы на первом этаже, вам, скорее всего, надо вверх. Если на последнем — вниз. В качестве домашнего задания продумайте, куда можно ехать с какого-то из средних этажей. Когда вы разберетесь с тем, где вы находитесь и куда вам нужно, напротив соответствующего этажа появится треугольничек, обращенный вершиной вверх или вниз в зависимости от направления движения лифта. По мере возникновения других запросов рядом с номерами этажей будут появляться другие треугольнички.

Если лифт находится там же, где и вы, его двери немедленно откроются. Что касается программы, то перед вашими глазами возникнет счастливое лицо офисного работника, заходящего вовнутрь. В противном случае лифт начнет двигаться вверх или вниз, пока не достигнет этажа, с которого был сделан запрос.

Ввод конечного пункта путешествия

Пока счастливый пассажир находится внутри лифта, нужно быстренько поинтересоваться у него, куда это он собрался ехать:

```
Лифт 1 находится на этаже 1.
Введите номера нужных этажей (0 для окончания ввода)
Этаж назначения 1: 13
```

Пассажир ввел 13. Но пока он размышлял, куда он хочет прокатиться, пришли еще пассажиры и стали нажимать разные кнопочки в лифте, всем ведь нужно на разные этажи! Поэтому программа должна позволять ввести несколько номеров этажей. Введите несколько номеров (не больше 20 штук), нажмите 0 для окончания ввода.

Пункты назначения, указанные пассажирами, показываются в виде маленьких прямоугольничков слева от изображения лифта, напротив соответствующего этажа. Каждый лифт имеет свой набор пунктов назначения (в отличие от запросов с исходных этажей, общих для всех лифтов).

Запросов с разных этажей можно делать сколько угодно. Программа запомнит их, как и выбранные из каждого лифта этажи назначения, и будет пытаться обслужить всех. Все четыре лифта могут быть одновременно в движении. На рис. 13.5 показана ситуация с несколькими запросами с этажей и несколькими запросами пунктов назначения.

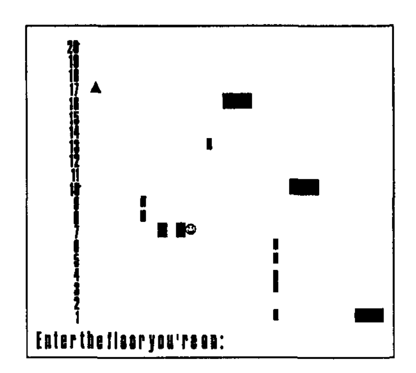


Рис. 13.5. Лифты в действии

Проектирование системы

Лифты могут быть приблизительно одинаковыми, поэтому есть смысл сделать их объектами одного класса под названием elevator. В этом классе содержатся данные, характерные для каждого отдельно взятого лифта: текущее местоположение, направление движения, номера этажей назначения и т. д.

Тем не менее есть данные обо всем здании в целом. Эти данные будут частью класса building. Во-первых, есть массив *запросов с этажей*. Это список этажей, где люди, ждущие лифта, нажали кнопку «вверх» или «вниз» и тем самым попросили лифт подъехать к ним. Любой из лифтов может отвечать на эти запросы, поэтому все они должны иметь доступ к этому массиву. В программе использу-

ется массив размером N×2 типа bool, где N — число этажей, а 2 поля на каждый этаж позволяют разделять запросы на движение вверх и вниз. Все лифты обращаются к этому массиву, чтобы понять, что им делать дальше.

Кроме того, каждый лифт должен быть осведомлен о том, где находятся остальные. Если один из лифтов находится на первом этаже, то нет никакого резона гнать его на пятнадцатый, если есть лифт, находящийся на десятом. Выполнять запросы должен ближайший к месту назначения лифт. Для обеспечения осведомления лифтов обо всех других в класс building вводится также массив указателей на лифты. Каждый объект (лифт) хранит свой адрес в памяти на момент создания, чтобы другие могли его отыскать.

Третий элемент данных в этом классе — это число уже созданных лифтов. Это позволяет лифтам иметь свой уникальный идентификатор.

Управление временем

В main() один из методов класса building вызывается через определенные интервалы времени, чтобы моделирование было более динамичным. Этот метод называется master_tick(). Он, в свою очередь, вызывает функцию для каждого лифта под названием car_tick1(). Она, кроме всего прочего, прорисовывает лифт на экране и вызывает еще одну функцию для принятия решения о том, что делать дальше. Выбор действий богат: поехать вниз, поехать вверх, остановиться, посадить пассажира, выпустить пассажира.

После этого каждый лифт должен быть сдвинут на свою новую позицию. Так. Что-то в этом месте все усложняется. Поскольку каждый лифт должен знать, где находятся остальные, прежде чем принять решение, все лифты должны пройти через процесс принятия решения, прежде чем кто-либо из них сдвинется с места. Чтобы удостовериться в том, что это действительно имеет место, мы запускаем tick'и дважды для каждой кабинки. Таким образом, car_tick1() вызывается для принятия решения о том, куда каждый лифт поедет, а другая функция — car_tick2() — для реального запуска лифтов. Изменяется значение переменной current_floor — кабинки начинают двигаться.

Процесс посадки пассажиров включает в себя последовательность определенных шагов, которые на экране отображаются так:

1. кабинка с закрытыми дверями, счастливое лицо отсутствует;
2. кабинка с открытыми дверями, счастливое лицо слева;
3. кабинка со счастливым лицом в открытых дверях, узнать от пассажира конечный этаж;
4. кабинка с закрытыми дверями, счастливое лицо отсутствует.

При высадке пассажира применяется обратный порядок. Эти последовательности действий осуществляются с помощью таймера (целочисленной переменной), который уменьшается с каждой временной отметкой (tick) от 3 до 0. С помощью case в функции car_display() выбирается и рисуется на экране соответствующая версия изображения лифта.

Поскольку программа ELEV использует функции консольной графики, должен быть доступ к заголовочным файлам библиотек MSOFTCON.H для компилято-

ров Microsoft или BORLACON.H для компиляторов Borland (см. приложение Д «Упрощенный вариант консольной графики»).

Листинг программы ELEV

Мы разделили программу на четыре файла. Два из них, ELEV.H и ELEV.CPP, могут поставляться, например, производителем программного обеспечения лифтов. Затем это ПО может купить строительная компания, заинтересованная в проектировании лифтовой системы здания. (Приводимая здесь программа не имеет надлежащего сертификата, поэтому лучше не пытаться использовать ее для работы с реальными лифтами.) Затем инженеры могут написать другую пару файлов, ELEV_APP.H и ELEV_APP.CPP, в первом из которых описаны характеристики высотного здания. Действительно нужно использовать разные файлы, потому что эти характеристики должны быть доступны методам класса лифтов, и простейшим решением для этого является именно включение ELEV_APP.H в ELEV.H. В файле ELEV_APP.CPP инициализируются лифты, затем в определенные моменты времени вызываются функции работы лифтов, что создает более точную картину в реальном времени.

Описатель класса

В файле ELEV.H содержится описатель класса elevator. Массив указателей на лифты, car_list[], позволяет каждой кабинке опрашивать все другие, выясняя их местонахождение и направление движения.

Листинг 13.4. Заголовочный файл ELEV.H

```
// elev.h
// заголовочный файл для лифтов - содержит объявления  //классов

#include "elev_app.h"  //поставляется клиентом
                       //(застройщиком)
#include "msoftcon.h"  //для консольной графики
#include <iostream>
#include <iomanip>    //для setw()
#include <conio.h>     //для вывода на экран
#include <stdlib.h>    //для itoa()
#include <process.h>    //для exit()
using namespace std;

enum direction { UP, DN, STOP };
const int LOAD_TIME =   3;   //время посадки/высадки
                             //(В tick'ах)
const int SPACING =    7;    //расстояние между
                             //изображениями кабинок
const int BUF_LENGTH =  80;  //длина буфера рабочей строки

class building;        //будущее объявление
////////////////////////////////////////////////////////////////
class elevator
  {
  private:
  building* ptrBuilding;  //указатель на building
```

продолжение

Листинг 13.4 (*продолжение*)

```
  const int car_number;   //номер лифта (от 0 до nc-1)
  int current_floor;      //где мы? (от 0 до nf-1)
  int old_floor;          //куда едем? (от 0 до nf-1)
  direction current_dir;         //в каком направлении?
  bool destination[NUM_FLOORS]:  //выбирается пассажирами
  int loading_timer;      //ненулевое во время посадки
  int unloading_timer;    //ненулевое во время высадки

  public:
  elevator(building*, int);  //конструктор
  void car_tick1();          //метроном-1 для каждой кабинки
  void car_tick2();          //метроном-2 для каждой кабинки
  void car_display();        //рисование лифта
  void dests_display() const;//вывод запросов
  void decide();         //принятие решения
  void move();           //движение лифта
  void get_destinations();  //получение номеров конечных
                            //этажей
  int get_floor() const;           //получение текущего этажа
  direction get_direction() const;  //получение текущего
//направления
  };
///////////////////////////////////////////////////////////
class building
  {
  private:
  elevator* car_list[NUM_CARS];  //указатель на кабинки
  int num_cars;                  //уже созданные лифты
                                 //массив кнопок «вверх», «вниз»
  bool floor_request[2][NUM_FLOORS];//false=вверх, true=вниз

  public:
  building();              //конструктор
  ~building();             //деструктор
  void master_tick();      //рассылка временных меток всем
                           //лифтам
  int get_cars_floor(const int) const; //поиск лифта
          //выяснение направления движения
  direction get_cars_dir(const int) const;
          //проверка запроса с этажа
  bool get_floor_req(const int, const int) const;
          //установка запроса с этажа
  void set_floor_req(const int, const int, const bool);
  void record_floor_reqs();  //получение запроса с этажа
  void show_floor_reqs() const;  //вывод запроса
  };
///////////////////////////////////////////////////////////
```

Методы

В файле ELEV.CPP содержатся определения класса elevator и методы и данные класса building. Функции building инициализируют систему, устанавливают метроном, получают и выводят запросы. Функции elevator инициализируют с помощью конструктора отдельные лифты, устанавливают два метронома для каждой кабинки, выводят изображение лифтов, их направления, принимают решения, двигают лифты и получают номера конечных этажей от пользователя.

Листинг 13.5. Листинг определений классов

```
// elev.cpp
// содержит определения данных и методов класса

#include "elev.h"               //включить объявление класса
//////////////////////////////////////////////////////////
//       определения функций для класса building
//////////////////////////////////////////////////////////
building::building()            //конструктор
  {
  char ustring[BUF_LENGTH];     //строка для номеров этажей

  init_graphics();              //инициализация графики
  clear_screen();               //очистка экрана
  num_cars = 0;
  for(int k=0; k<NUM_CARS; k++) //создать лифты
    {
    car_list[k] = new elevator(this, num_cars);
    num_cars++;
    }
  for(int j=0; j<NUM_FLOORS; j++)    //для каждого этажа
    {
    set_cursor_pos(3, NUM_FLOORS-j); //вывести номер этажа
    itoa(j+1, ustring, 10);          //на экран
    cout << setw(3) << ustring;
    floor_request[UP][j] = false;    //запросов еще не было
    floor_request[DN][j] = false;
    }
  } //конец конструктора
//--------------------------------------------------------
building::~building()           //деструктор
  {
  for(int k=0; k<NUM_CARS; k++) //удалить лифты
    delete car_list[k];
  }
//--------------------------------------------------------
void building::master_tick()    //метроном
  {
  int j;
  show_floor_reqs();            //вывести запросы с этажей
  for(j=0; j<NUM_CARS; j++)     //для каждого лифта
    car_list[j]->car_tick1();   //послать временную метку 1
  for(j=0; j<NUM_CARS; j++)     //для каждого лифта
    car_list[j]->car_tick2();   //послать временную метку 2
  } //конец master_tick()
//--------------------------------------------------------
void building::show_floor_reqs() const //вывод запросов
  {
  for(int j=0; j<NUM_FLOORS; j++)
    {
    set_cursor_pos(SPACING, NUM_FLOORS-j);
    if(floor_request[UP][j]==true)
      cout << '\x1E';           //стрелка вверх
    else
      cout << ' ';
    set_cursor_pos(SPACING+3, NUM_FLOORS-j);
```

продолжение

Листинг 13.5 (*продолжение*)

```
    if(floor_request[DN][j]==true)
      cout << '\x1F';          //стрелка вниз
    else
      cout << ' ';
    }
  }  //конец show_floor_reqs()
//--------------------------------------------------------------
//record_floor_reqs() - получение запросов снаружи
void building::record_floor_reqs()
  {
  char ch = 'x';                //рабочий символ для ввода
  char ustring[BUF_LENGTH];     //рабочая строка ввода
  int iFloor;           //этаж, с которого был запрос
  char chDirection;     //'u' или 'd' для выбора направления

  set_cursor_pos(1,22);         //низ экрана
  cout << "Нажмите [Enter] для вызова лифта: ";
  if( !kbhit() )                //ожидание нажатия (должен быть CR,
                                //возврат каретки)
    return;
  cin.ignore(10, '\n');
  if(ch=='\x1B')      //при нажатии escape выход из программы
    exit(0);
  set_cursor_pos(1,22); clear_line();  //очистить от старого
                                       //текста
  set_cursor_pos(1,22);         //низ экрана
  cout << "На каком Вы этаже? ";
  cin.get(ustring, BUF_LENGTH); //получить этаж
  cin.ignore(10, '\n');         //съесть символы, включая
                                //разделитель
  iFloor = atoi(ustring);       //преобразовать в int

  cout << "В каком направлении будете двигаться (u/d): ";
  cin.get(chDirection);         //(избежать повтора новых
                                //строк)
  cin.ignore(10, '\n');         //съесть символы, включая разделитель

  if(chDirection=='u' || chDirection=='U')
    floor_request[UP][iFloor-1] = true;  //запрос «вверх»
  if(chDirection=='d' || chDirection=='D')
    floor_request[DN][iFloor-1] = true;  //запрос «вниз»
  set_cursor_pos(1,22); clear_line();//стереть старый текст
  set_cursor_pos(1,23); clear_line();
  set_cursor_pos(1,24); clear_line();
  }  //end record_floor_reqs()
//--------------------------------------------------------------
//get_floor_req() - проверка наличия запросов
bool building::get_floor_req(const int dir,
                    const int floor) const
  {
  return floor_request[dir][floor];
  }
//--------------------------------------------------------------
//set_floor_req() - установить запрос
void building::set_floor_req(const int dir, const int floor,
                  const bool updown)
```

```
  {
  floor_request[dir][floor] = updown;
  }
//------------------------------------------------------------
//get_cars_floor() - найти кабинку
int building::get_cars_floor(const int carNo) const
  {
  return car_list[carNo]->get_floor();
  }
//------------------------------------------------------------
//get_cars_dir() - определение направления
direction building::get_cars_dir(const int carNo) const
  {
  return car_list[carNo]->get_direction();
  }
//------------------------------------------------------------

////////////////////////////////////////////////////////////
//        определения функций класса elevator
////////////////////////////////////////////////////////////
                                //конструктор
elevator::elevator(building* ptrB, int nc) :
                     ptrBuilding(ptrB), car_number(nc)
  {
  current_floor = 0;            //начать с 0 (для пользователя - 1)
  old_floor = 0;                //запомнить предыдущий этаж
  current_dir = STOP;           //вначале стоит на месте
  for(int j=0; j<NUM_FLOORS; j++)//пассажиры еще не
    destination[j] = false;      //  нажимали кнопки
  loading_timer = 0;            //еще не началась посадка
  unloading_timer = 0;          //еще не началась высадка
                      }         //конец конструктора
//------------------------------------------------------------
int elevator::get_floor() const//получение текущего этажа
  {
  return current_floor;
  }
//------------------------------------------------------------
direction elevator::get_direction() const  //получение
  {                                        //текущего направления
  return current_dir;
  }
//------------------------------------------------------------
void elevator::car_tick1()      //метроном-1 для каждого лифта
  {
  car_display();                //нарисовать кабинку
  dests_display();              //нарисовать конечные этажи
  if(loading_timer)             //счет времени посадки
    --loading_timer;
  if(unloading_timer)           //счет времени высадки
    --unloading_timer;
  decide();                     //принятие решения
  }  //конец car_tick()
//------------------------------------------------------------
//все лифты должны знать, что делать, до начала движения
void elevator::car_tick2()      //метроном-2 для каждого лифта
  {
```

продолжение ⇨

Листинг 13.5 (*продолжение*)

```
  move();                       //двигать лифт, если нужно
  }
//---------------------------------------------------------
void elevator::car_display()    //вывод образа лифта
  {
  set_cursor_pos(SPACING+(car_number+1)*SPACING, NUM_FLOORS-old_floor);
  cout << "   ";                //стереть со старой позиции
  set_cursor_pos(SPACING-1+(car_number+1)*SPACING,
                        NUM_FLOORS-current_floor);
    switch(loading_timer)
    {
      case 3:
      cout << "\x01\xDB \xDB "; //лифт с открытыми дверями
      break;                    //слева - мордочка
    case 2:
      cout << " \xDB\x01\xDB "; //мордочка в дверях
      get_destinations();       //получить конечный этаж
      break;
    case 1:
      cout << " \xDB\xDB\xDB "; //нарисовать с закрытыми
      break;                    //дверями без мордочки
    case 0:
      cout << " \xDB\xDB\xDB ";  //двери закрыты,
      break;                     //мордочки нет (по умолчанию)
    }
  set_cursor_pos(SPACING+(car_number+1)*SPACING,
                        NUM_FLOORS-current_floor);
  switch(unloading_timer)
    {
    case 3:
      cout << "\xDB\x01\xDB ";  //двери открыты,
      break;                    //мордочка слева
    case 2:
      cout << "\xDB \xDB\x01";  //двери открыты,
      break;                    //мордочка справа
    case 1:
      cout << "\xDB\xDB\xDB ";  //двери закрыты
      break;                    //мордочки нет
    case 0:
      cout << "\xDB\xDB\xDB ";  //двери закрыты,
      break;                    //мордочки нет (по умолчанию)
    }
  old_floor = current_floor;    //запомнить старый этаж
  }  //конец car_display()
//---------------------------------------------------------
void elevator::dests_display() const //вывести конечные
  {      //  этажи, выбранные кнопками
  for(int j=0; j<NUM_FLOORS; j++)    //  внутри лифта
    {
    set_cursor_pos(SPACING-2+(car_number+1)*SPACING, NUM_FLOORS-j);
    if( destination[j] == true )
      cout << '\xFE';                //маленький квадратик
    else
      cout << ' ';                   //пробел
    }
  }  //конец dests_display()
```

```
//--------------------------------------------------------
void elevator::decide()          //принятие решения
  {
  int j;
//флаги показывают, сверху или снизу запросы или назначения
  bool destins_above, destins_below;   //конечные пункты
  bool requests_above, requests_below; //запросы
  //ближайший запрос снизу и сверху
  int nearest_higher_req = 0;
  int nearest_lower_req = 0;
  //флаги указывают, есть ли другие лифты, движущиеся в том
  //же направлении между нами и ближайшим запросом с этажа //(ЗЭ)
  bool car_between_up, car_between_dn;
  //флаги указывают, есть ли лифты противоположного
  // направления с другой стороны от ближайшего ЗЭ
  bool car_opposite_up, car_opposite_dn;
  //этаж и направление другого лифта
  int ofloor;                     //этаж
  direction odir;                 //направление

  //убедиться, что мы не слишком высоко или низко
  if( (current_floor==NUM_FLOORS-1 && current_dir==UP)
    || (current_floor==0 && current_dir==DN) )
    current_dir = STOP;

  //если этаж назначения - текущий, начать высадку
  if( destination[current_floor]==true )
    {
    destination[current_floor] = false; //удалить это
                                        / назначение
    if( !unloading_timer)       //высадка
      unloading_timer = LOAD_TIME;
    return;
    }
  //если есть запрос «вверх» с этого этажа и если
  //мы едем вверх или стоим, произвести посадку на борт
  if( (ptrBuilding->get_floor_req(UP, current_floor) &&
      current_dir != DN) )
    {
    current_dir = UP;           //(если была остановка)
    //удалить ЗЭ в данном направлении движения
    ptrBuilding->set_floor_req(current_dir,
                          current_floor, false);
    if( !loading_timer)         //посадка
      loading_timer = LOAD_TIME;
    return;
    }
  //Если запрос «вниз» с этого этажа и если мы едем вниз или
  //стоим, посадить пассажиров
  if( (ptrBuilding->get_floor_req(DN, current_floor) &&
      current_dir != UP) )
    {
    current_dir = DN;           //(если была остановка)
    //удалить ЗЭ данного направления
    ptrBuilding->set_floor_req(current_dir,
                           current_floor, false);
    if( !loading_timer)         //посадка
```

продолжение

Листинг 13.5 (*продолжение*)

```
      loading_timer = LOAD_TIME;
    return;
    }
  //проверка других назначений или ЗЭ
  //расстояние считать до ближайшего запроса
  destins_above = destins_below = false;
  requests_above = requests_below = false;
  for(j=current_floor+1; j<NUM_FLOORS; j++)
    {                            //проверять верхние этажи
    if( destination[j] )         //если они - назначения
      destins_above = true;      //установить флаг
    if( ptrBuilding->get_floor_req(UP, j) ||
       ptrBuilding->get_floor_req(DN, j) )
      {                          //если ЗЭ
      requests_above = true;     //установить флаг
      if( !nearest_higher_req ) //если еще не установлен
        nearest_higher_req = j; //установить ближайший ЗЭ
      }
    }
  for(j=current_floor-1; j>=0; j--) //проверка нижних этажей
    {
    if(destination[j] )          //если назначения
      destins_below = true;      //установить флаг
    if( ptrBuilding->get_floor_req(UP, j) ||
       ptrBuilding->get_floor_req(DN, j) )
      {                          //если запросы
      requests_below = true;     //установить флаг
      if( !nearest_lower_req )  //если еще не установлен
        nearest_lower_req = j;  //установить ближайший ЗЭ
      }
    }
  //если нет запросов сверху/снизу, остановиться
  if( !destins_above && !requests_above &&
     !destins_below && !requests_below)
     {
     current_dir = STOP;
     return;
     }
  //если есть назначение, а мы стоим или движемся к нему,
  //начать/продолжать движение
  if( destins_above && (current_dir==STOP || current_dir==UP) )
    {
    current_dir = UP;
    return;
    }
  if( destins_below && (current_dir==STOP || current_dir==DN) )
    {
    current_dir = DN;
    return;
    }
  //проверка, есть ли другие лифты, (a) того же направления
  //между нами и ближайшим ЗЭ, или
  //(b) встречного направления с другой стороны ЗЭ
  car_between_up = car_between_dn = false;
  car_opposite_up = car_opposite_dn = false;

  for(j=0; j<NUM_CARS; j++)      //проверить каждый лифт
```

```
{
if(j != car_number)          //если это не наш лифт
  {                          //получить его этаж
  ofloor = ptrBuilding->get_cars_floor(j):  //и
  odir = ptrBuilding->get_cars_dir(j): //направление

  //если едет вверх. и ЗЭ вверху
  if( (odir==UP || odir==STOP) && requests_above )
 //если он при этом между нами и ЗЭ
    if( (ofloor > current_floor
       && ofloor <= nearest_higher_req)
    //или там же. но его номер ниже
      || (ofloor==current_floor && j < car_number) )
      car_between_up = true;
  //если он едет вниз. и ЗЭ внизу
  if( (odir==DN || odir==STOP) && requests_below )
    //если он снизу. но над ближайшим ЗЭ
    if( (ofloor < current_floor
       && ofloor >= nearest_lower_req)
      //или на том же этаже. но с меньшим номером
      || (ofloor==current_floor && j < car_number))
      car_between_dn = true:
  //если идет вверх. а ЗЭ снизу
  if( (odir==UP || odir==STOP) && requests_below )
    //и он ниже ЗЭ и ближе к нему. чем мы
    if(nearest_lower_req >= ofloor
      && nearest_lower_req - ofloor
        < current_floor - nearest_lower_req)
      car_opposite_up = true:
  //если идет вниз. а ЗЭ сверху
  if( (odir==DN || odir==STOP) && requests_above )
    //и он над ЗЭ и ближе к нему. чем мы
    if(ofloor >= nearest_higher_req
      && ofloor - nearest_higher_req
        < nearest_higher_req - current_floor)
      car_opposite_dn = true:
  }  //конец if(не для нашего лифта)
}  //конец for(для каждого лифта)

//если идем вверх или остановились. а ЗЭ над нами
//и между нами и ЗЭ нет идущих вверх лифтов
//или идущих вниз над ЗЭ и ближе к нему. чем мы. тогда
//ехать вверх
if( (current_dir==UP || current_dir==STOP)
   && requests_above && !car_between_up && !car_opposite_dn )
  {
  current_dir = UP;
  return;
  }

//если идем вниз или остановились. и снизу есть ЗЭ.
//и нет лифтов. идущих вниз. между нами и ЗЭ
//или под ЗЭ. идущих вверх и ближе нас к ЗЭ
if( (current_dir==DN || current_dir==STOP)
   && requests_below && !car_between_dn && !car_opposite_up )
  {
  current_dir = DN;
  return:
```

продолжение

Листинг 13.5 (*продолжение*)

```
    }
  //если больше ничего не происходит, остановиться
  current_dir = STOP;
  } //end decide(), finally
//------------------------------------------------------------
void elevator::move()
  {                              //если посадка или высадка,
  if(loading_timer || unloading_timer) //не двигаться
    return;
  if(current_dir==UP)            //если идем вверх, идти вверх
    current_floor++;
  else if(current_dir==DN)       //если идем вниз, идти вниз
    current_floor--;
  } //end move()
//------------------------------------------------------------
void elevator::get_destinations() //остановка, получение
                                  //пунктов назначения
  {
  char ustring[BUF_LENGTH];       //входной рабочий буфер
  int dest_floor;                 //этаж назначения

  set_cursor_pos(1,22);clear_line();//удалить верхнюю строку
  set_cursor_pos(1, 22);
  cout << "Лифт " << (car_number+1)
      << " остановился на этаже " << (current_floor+1)
      << "\nЭтаж назначения (0 для окончания ввода)";
  for(int j=1; j<NUM_FLOORS; j++)   //получить запросы этажей
    {             //максимум; обычно меньше
    set_cursor_pos(1, 24);
    cout << "Этаж назначения " << j << ": ";

    cin.get(ustring, BUF_LENGTH);   //(во избежание дублирования
                                    //пустых строк)
    cin.ignore(10, '\n');        //съесть символы, включая
                                 //ограничитель
    dest_floor = atoi(ustring);
    set_cursor_pos(1,24); clear_line(); //стереть старую
                                        //строку
    if(dest_floor==0)            //если больше нет запросов
      {         //стереть нижние три строки
      set_cursor_pos(1,22); clear_line();
      set_cursor_pos(1,23); clear_line();
      set_cursor_pos(1,24); clear_line();
      return;
      }
    --dest_floor;                   //начинать с 0, а не 1
    if(dest_floor==current_floor)   //выбрать текущий этаж
      { --j; continue; }            //  забыть его
    //если мы остановились, первый запрос выбирает
                                    //направление движения
    if(j==1 && current_dir==STOP)
      current_dir = (dest_floor < current_floor) ? DN : UP;
    destination[dest_floor] = true;//записать выбор
    dests_display();                //вывести этажи назначения
    }
  } //конец get_destinations()
//------------------------------------------------------------
```

Прикладная программа

Следующие два файла, ELEV_APP.H и ELEV_APP.CPP, созданы некими проектировщиками лифтовой системы конкретного здания. Соответственно, им нужно адаптировать ПО для своих целей. Для этого в файле ELEV_APP.H введены две константы — число этажей и число лифтов.

Листинг 13.6. Заголовочный файл ELEV_APP.H

```
// elev_app.h
// Устанавливает характеристики конкретного здания

const int NUM_FLOORS = 20;   //число этажей
const int NUM_CARS = 4;      //число кабинок лифтов
```

В файле ELEV_APP.CPP инициализируются данные класса building и создается набор объектов типа elevator с помощью new (впрочем, можно было использовать и массив). Затем в цикле вызываются методы master_tick() и get_floor_request(). Функция wait() (объявленная в MSOFTCON.H или BORLACON.H) замедляет процесс для удобства восприятия человеком. Когда пользователь отвечает на запрос программы, таймер (время в программе, в противоположность пользовательскому времени) останавливается.

Листинг 13.7. Программа ELEV_APP

```
// elev_app.cpp
// клиентский файл

#include "elev.h"              //для объявлений классов

int main()
  {
  building theBuilding;
  while(true)
    {
    theBuilding.master_tick(); //послать временные метки
                               //всем лифтам
    wait(1000);                //пауза
                //получить запросы этажей от пользователей
    theBuilding.record_floor_reqs();
    }
  return 0;
  }
```

Стратегия работы лифтов

Встраивание интеллекта необходимого уровня в лифтовые системы — не такая простая задача, как кажется на первый взгляд. Процесс принятия решения отрабатывается функцией decide(), состоящей из определенного набора правил. Эти правила организованы в порядке их приоритета. Если применяется одно из них, то выполняется некое действие, при этом правила более низких уровней не отрабатываются. Приведем упрощенный вариант такой обработки:

1. Если лифт вот-вот врежется в дно шахты или пробьет ее крышу, наверное, следует остановиться.

2. Если данный этаж — это этаж назначения, высадить пассажиров.
3. Если обнаружен на данном этаже запрос «вверх», едем вверх, загружаем пассажиров.
4. Если обнаружен на данном этаже запрос «вниз», едем вниз, загружаем пассажиров.
5. Если нет запросов этажей или этажей назначения ни снизу, ни сверху, остановиться.
6. Если этажи назначения сверху, едем вверх.
7. Если этажи назначения снизу, едем вниз.
8. Если стоим или движемся вверх, есть запрос с более высокого этажа, и нет при этом лифтов, движущихся вверх, между нами и этажом запроса или над ним, а также движущихся вниз и находящихся ближе к нему, чем мы, то едем вверх.
9. Если стоим или движемся вниз, есть запрос с более низкого этажа, и нет при этом лифтов, движущихся вниз, между нами и этажом запроса или под ним, а также движущихся вверх и находящихся ближе к нему, чем мы, то едем вниз.
10. Если ничего никому от нас не нужно, останавливаемся.

Правила 8 и 9 довольно сложны. Они предназначены для исключения выполнения одного и того же запроса несколькими лифтами сразу. Тем не менее результаты не всегда идеальны. В некоторых ситуациях лифты медлят с принятием решения, очень опасаясь сделать то, что могут сделать за них другие. Но в реальности в этот момент другие лифты не движутся к той же цели, а, например, отвечают на свои запросы. Чтобы улучшить стратегию работы системы, необходимо заставить различать функцию decide() запросы «вверх» и «вниз» во время проверки относительного местонахождения ЗЭ (запроса с этажа). Но это еще больше усложнило бы и без того слишком длинную функцию, поэтому оставляем возможность дальнейших усовершенствований читателю.

Диаграмма состояний для программы ELEV

В главе 10 «Указатели» мы представляли диаграмму состояний UML. Давайте теперь взглянем на диаграмму состояний для нашей программы моделирования лифтов. Для того чтобы немного упростить задачу, будем считать, что в здании находится только один человек и имеется только один лифт. Таким образом, в один момент времени может быть только один запрос этажа и один этаж назначения. Лифту не нужно следить за другими. Диаграмма состояний представлена на рис. 13.6.

Обозначения в диаграмме таковы: «cd» — этаж назначения, то есть кнопка, нажатая внутри лифта (грубо говоря, это соответствует массиву destination из нашей программы); «fr» — запрос с этажа, кнопка, нажатая снаружи лифта, грубо говоря, значение переменной floor_req.

Состояния лифта определяются по значениям переменной current_dir и состояниям loading_timer и unloading_timer. Поскольку все состояния переходят друг в друга с помощью слова «вдруг» (то есть по сигналам таймера), на диаграмме показаны только контрольные состояния. Отображаются представления лифта о запросах с этажей и этажей назначения.

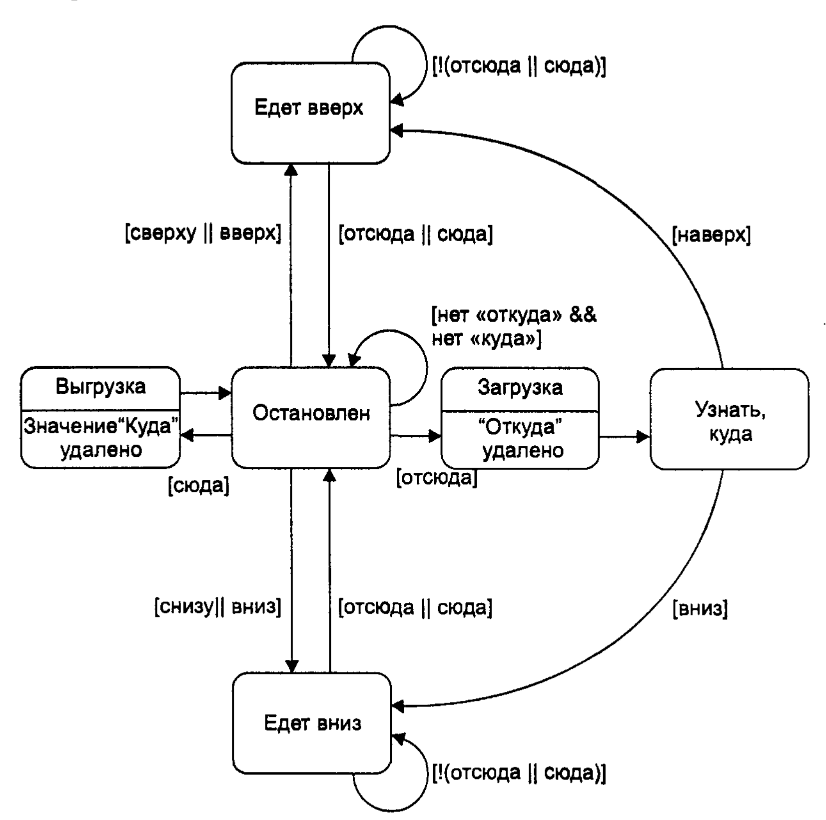


Рис. 13.6. Диаграмма состояний объекта elevator

Резюме

Поставляемые производителем программного обеспечения библиотеки классов чаще всего состоят из двух частей: общедоступного компонента (интерфейса), содержащего объявления классов в заголовочном файле, и скрытого компонента (реализации), содержащего определения методов в объектном (.OBJ) или библиотечном (.LIB) файле.

Компиляторы С++ позволяют соединять разнообразные исходные и объектные файлы в единую исполняемую программу. Благодаря этому можно воспользоваться библиотеками функций даже разных производителей, чтобы с использованием их классов создать в итоге хорошее приложение. Файл проек-

та позволяет запоминать, что и когда было откомпилировано, чтобы не проделывать эту процедуру заново для всех без исключения файлов, входящих в проект. Можно компилировать только те исходные файлы, которые изменялись со времени последней сборки.

Межфайловая коммуникация подразумевает, что переменные, функции и объекты определены в одном файле, а объявлены в другом — в том, где они реально используются. Определение класса должно быть помещено во все файлы, которые обращаются к нему. Нужно внимательно следить за тем, чтобы не возникало повторных определений как в исходных, так и в заголовочных файлах.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Разбивать программу на несколько файлов желательно, потому что:
   а) некоторые файлы не нуждаются в перекомпиляции при каждой сборке;
   б) программа может быть разделена на функциональные элементы;
   в) файлы можно представлять на рынке в объектном виде;
   г) разные программисты могут работать над разными файлами.
2. Заголовочный файл связывается с исходным с помощью ____________.
3. Объектный файл присоединяется к исходному с помощью ____________.
4. В файл проекта включаются:
   а) данные о содержимом файлов, входящих в проект;
   б) даты последних изменений файлов, входящих в проект;
   в) команды компиляции и компоновки;
   г) определения переменных C++.
5. Группу связанных между собой классов, поставляющуюся в виде отдельного программного продукта, часто называют ____________________.
6. Истинно ли утверждение о том, что заголовочный файл может нуждаться в доступе со стороны нескольких исходных файлов в проекте?
7. Так называемые скрытые файлы библиотеки классов:
   а) требуют пароль;
   б) могут быть сделаны доступными с помощью дружественных функций;
   в) помогают защитить код от пиратов;
   г) могут состоять только из объектного кода.
8. Истинно ли утверждение о том, что библиотеки классов — это более мощный инструмент, чем библиотеки функций?
9. Истинно ли утверждение о том, что интерфейс является скрытой частью библиотеки, а реализация — общедоступной?

10. Общедоступная часть библиотеки классов обычно содержит:
    а) объявления методов;
    б) определения методов;
    в) объявления классов;
    г) определения небольших функций.
11. Два или более исходных файла могут быть соединены путем их _______.
12. Истинно ли утверждение о том, что переменная, определенная внутри функции, видна на протяжении всего исходного текста того файла, в котором она определена?
13. Глобальная переменная определена в файле А. Чтобы получить доступ к ней из файла В, необходимо:
    а) определить ее в файле В, используя extern;
    б) определить ее в файле В, используя static;
    в) расслабиться (ничего не надо делать);
    г) объявить ее в файле В, используя extern.
14. Часть программы, в которой к данной переменной имеют доступ переменные из других частей программы, называется ______________________.
15. Файлы, которые реально соединяются компоновщиком, называются ____ ______________________________.
16. Функция определена в файле А. Для того чтобы можно было вызывать ее из файла В, функция должна быть ____________ в ______________.
17. Истинно ли утверждение о том, что объявление функции не требует использования extern?
18. Чтобы определять объекты класса в разных файлах, в каждом из них необходимо:
    а) объявлять класс;
    б) определять класс;
    в) объявлять класс с использованием extern;
    г) определять класс с использованием extern.
19. Истинно ли утверждение о том, что переменная, определенная в заголовочном файле, может быть доступна из двух исходных файлов, если в каждый из них включен этот заголовочный файл?
20. Конструкция #if !defined()...#endif может использоваться для предотвращения повторных определений, когда:
    а) два заголовочных файла включены в исходный файл;
    б) заголовочный файл включен в два исходных;
    в) два заголовочных файла включены в два исходных;
    г) один заголовочный файл включен в другой, и оба они включены в исходный.

21. Пространства имен используются для:
    а) автоматизации именования переменных;
    б) сужения области видимости элементов программы;
    в) разделения программы на отдельные файлы;
    г) предотвращения использования длинных имен переменных.
22. Для определения пространства имен используется формат, сходный с определением класса, но вместо зарезервированного слова class используется ________________.
23. Использование typedef позволяет:
    а) сокращать длинные имена переменных;
    б) менять имена типов;
    в) сокращать длинные названия функций;
    г) менять названия классов.

Проекты

К сожалению, объем книги не позволяет описать упражнения такого типа и масштаба, как примеры, приведенные в этой главе. И все же предлагаем несколько идей различных проектов, которые вы можете развить самостоятельно.

1. Создайте методы для вычитания и деления чисел класса verylong (см. одноименный пример). Это будут перегружаемые операции «-» и «/». Здесь потребуется некоторое напряжение интеллекта. Будем предполагать при написании функции вычитания, что числа могут быть как положительными, так и отрицательными. Это усложнит умножение и сложение, которым придется выполнять различные действия в зависимости от знака числа.

   Чтобы вникнуть в алгоритм, прежде всего попробуйте выполнить деление больших чисел вручную, на бумажке, записывая каждый шаг. Затем внедрите все эти шаги в новый метод класса verylong. При этом обнаружится, что приходится выполнять ряд операций сравнения, так что необходимо будет написать специальную функцию еще и для этого.
2. Перепишите программу ELEV таким образом, чтобы она работала только с одним лифтом. Это сильно упростит программное решение. Удалите ненужные части программы. Кроме того, можно представить себе задачу с одним лифтом и одним пассажиром, что еще больше упростит дело.
3. Модифицируйте программу работы с лифтами, улучшив методы обработки запросов. Чтобы увидеть огрехи в поведении нынешнего варианта программы, попробуйте создать запрос «вниз» с двадцатого этажа. Затем создайте запрос «вниз» с десятого этажа. Лифт-1 тотчас поднимется на 20 этаж. Но лифт-2, который, по идее, должен подниматься на 10 этаж,

будет ждать, пока лифт-1 пройдет 10 этаж, только после этого начнет движение. Измените функцию decide() таким образом, чтобы исключить эту ошибку.

4. Создайте библиотеку классов, моделирующую какую-нибудь интересующую вас предметную область. Создайте main() или клиентскую программу для ее тестирования. Предложите свою библиотеку классов на рынке, станьте богатым и знаменитым.

Глава 14
Шаблоны и исключения

- Шаблоны функций
- Шаблоны классов
- Исключения

В этой главе рассматриваются два важных инструмента C++ — шаблоны и исключения. Шаблоны позволяют использовать одни и те же функции или классы для обработки данных разных типов. Исключения дают возможность обрабатывать ошибки, возникающие в классах, удобным и универсальным способом. Эти две темы объединены в одной главе скорее по историческим причинам: эти два инструмента были введены в состав языка C++ практически одновременно. Они не входили в исходную спецификацию языка, но были представлены в виде «экспериментальных» в книге Эллиса и Страуструпа (Ellis and Stroustrup, 1990, см. приложение З «Библиография»), но постепенно они стали частью стандартного C++.

Концепция шаблонов может быть использована в двух видах: по отношению к функциям и по отношению к классам. Сначала мы взглянем на шаблоны функций, затем на шаблоны классов, а закончим главу разговором о механизме исключений.

Шаблоны функций

Допустим, вам нужно написать функцию вычисления модуля чисел. Из школьного курса алгебры вы, конечно, знаете, что модуль — это абсолютное значение числа, то есть число без знака. Напоминаем: модуль числа 3 равен 3, модуль числа −3 равен 3.

Скорее всего, функция вычисления модуля будет использоваться с каким-либо одним типом данных:

```
int abs(int n)              //вычисление модулей целых чисел
  {
  return (n<0) ? -n : n;  //если отрицательное, вернуть -n
  }
```

Определенная выше функция берет аргумент типа int и возвращает результат того же типа. Но теперь представьте, что понадобилось найти модуль числа типа long. Придется писать еще одну функцию:

```
long abs(long n)          //вычисление модулей чисел типа long
  {
  return (n<0) ? -n : n;
  }
```

Да, теперь еще одну для float:

```
float abs(float n)        //модуль целых чисел
  {
  return (n<0) ? -n : n;  //если отрицательное, вернуть -n
  }
```

Тело функций, как видите, ничем не отличается. И все же они совершенно разные, поскольку обрабатывают аргументы и возвращают значения разных типов. Они могут быть перегружены и иметь одинаковые имена, это правда, но все равно для каждой из них нужно писать отдельное определение. (Что касается языка С, в котором нет понятия перегрузки, нельзя использовать даже одинаковые имена для обозначения функций разных типов. Там это приведет к созданию целого клана функций с похожими названиями, таких, как abs(), fabs(), fabsl(), labs() и cabs().)

Многократное переписывание этих функций-близнецов очень утомляет, затрудняет читабельность листинга, зато делает его увесистым. Можно показать неопытному заказчику тома исходных кодов, демонстрируя свою напряженную работу над проектом. А если где-то в алгоритме попадется ошибка, придется исправлять ее в теле каждой функции. Причем некорректное исправление ошибки приведет к несостоятельности программы.

Было бы здорово, если бы был какой-то способ написать эту функцию всего один раз и заставить ее работать с различными типами данных, возвращая, соответственно, результаты разного типа. Это как раз то, для чего существуют шаблоны функций в C++. Эта простая и гениальная идея схематично изображена на рис. 14.1.

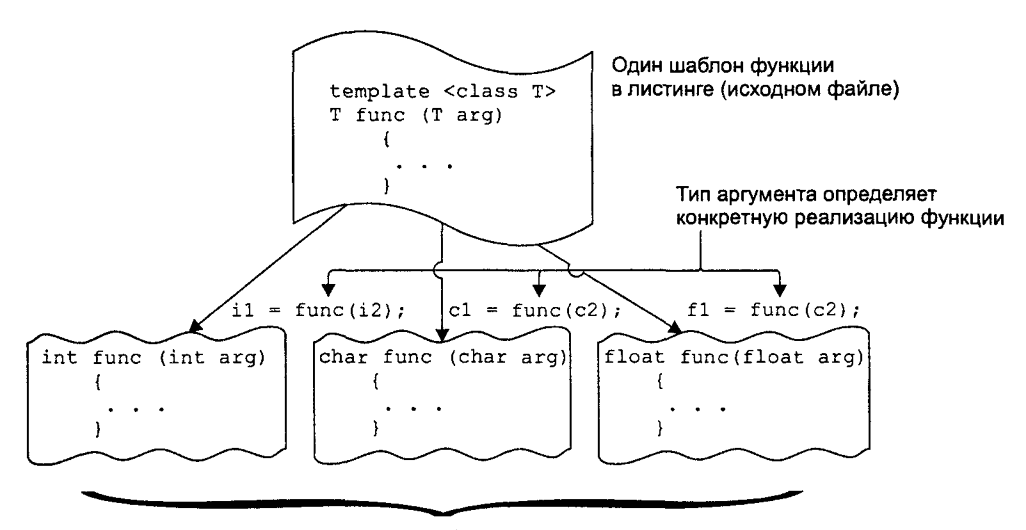


Рис. 14.1. Шаблон функции

Шаблон простой функции

Первый пример в этой главе показывает, как пишется шаблон функции, вычисляющей модуль числа. Шаблон работает со всеми базовыми числовыми типами. В программе сначала определяется шаблонная версия abs(), а затем в main() производится проверка правильности ее работы.

Листинг 14.1. Программа TEMPABS[1]

```
// tempabs.cpp
// Шаблон функции вычисления модуля числа
#include <iostream>
using namespace std;
//--------------------------------------------------------------
template <class T>           //Шаблон функции!
T abs(T n)
  {
  return (n < 0) ? -n : n;
  }
//--------------------------------------------------------------
int main()
  {
  int int1 = 5;
  int int2 = -6;
  long lon1 = 70000L;
  long lon2 = -80000L;
  double dub1 = 9.95;
  double dub2 = -10.15;
                             //осуществления вызовов
  cout << "\nabs(" << int1 << ")=" << abs(int1);  //abs(int)
  cout << "\nabs(" << int2 << ")=" << abs(int2);  //abs(int)
  cout << "\nabs(" << lon1 << ")=" << abs(lon1); //abs(long)
  cout << "\nabs(" << lon2 << ")=" << abs(lon2); //abs(long)
  cout << "\nabs(" << dub1 << ")=" << abs(dub1);
                                        //abs(double)
  cout << "\nabs(" << dub2 << ")=" << abs(dub2);
                                        //abs(double)
  cout << endl;
  return 0;
  }
```

Результаты работы программы:

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

Итак, теперь функция abs() может работать со всеми тремя типами данных (int, long, double). Типы определяются функцией при передаче аргумента. Она будет работать и с другими базовыми числовыми типами, да даже и с пользовательскими, лишь бы был надлежащим образом перегружен оператор «меньше» (<) и унарный оператор «–».

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

Вот спецификация функции abs() для работы с несколькими типами данных:

```
template <class T>          //шаблон функции
T abs (T n)
  {
  return (n<0) ? -n : n;
  }
```

Вся конструкция, начиная от ключевого слова template в начале первой строки и включая определение функции, называется *шаблоном функции*. Каким образом достигается такая потрясающая гибкость использования функции?

Синтаксис шаблона функции

Краеугольным камнем концепции шаблонов функции является представление использующегося функцией типа не в виде какого-либо специфического (например, int), а с помощью названия, вместо которого может быть подставлен *любой* тип. В приведенном примере таким именем является T (никакого сакрального смысла именно в таком названии нет. На его месте может все что угодно, например anyType или YooHoo). Ключевое слово template сообщает компилятору о том, что «сейчас будет море крови»: мы определяем шаблон функции. Ключевое слово class, заключенное в угловые скобки, можно с тем же успехом заменить на type. Как вы уже видели, можно определять собственные типы, используя классы, поэтому разницы между типами и классами в известном смысле нет совсем. Переменная, следующая за словом class (T в нашем примере), называется *аргументом шаблона*. Внутри всего определения шаблона любой конкретный тип данных, такой, как int, заменяет аргумент T. В abs() это имя встречается лишь дважды в первой строке в качестве типа аргумента и одновременно в качестве типа функции. В более сложном случае оно могло бы встретиться много раз, в том числе в теле функции.

Действия компилятора

Что компилятор делает, увидев ключевое слово template и следующее за ним определение функции? Правильно, почти ничего или очень мало. Дело в том, что сам по себе шаблон не вызывает генерацию компилятором какого-либо кода. Да и не может он этого сделать, поскольку еще не знает, с каким типом данных будет работать данная функция. Он разве что запоминает шаблон для будущего использования.

Генерации кода не происходит до тех пор, пока функция не будет реально вызвана в ходе выполнения программы. В примере TEMPABS это происходит в конструкциях типа abs(int) в выражении

```
cout << "\nabs(" << int1 << ")=" << abs(int1);
```

Когда компилятор видит такой вызов функции, он знает, что нужно использовать тип int, поскольку это тип аргумента int1, переданного в шаблон. Поэтому он генерирует код abs для типа int, подставляя его везде вместо T. Это называется *реализацией* шаблона функции; при этом каждый реализованный шаблон функции называется *шаблонной функцией*[1].

---

[1] То есть *шаблонная функция* — это реализация *шаблона функции*. О, великий, могучий, правдивый и свободный русский язык! — *Примеч. перев.*

Компилятор также генерирует вызов реализованной функции и вставляет его туда, где находится abs(int1). Выражение abs(lon1) приводит к тому, что компилятором генерируется версия abs(int), работающая с типом long, и ее вызов. То же самое касается типа float. Конечно, компилятор достаточно умен для того, чтобы создавать для каждого типа данных только одну копию abs(). Таким образом, несмотря на два вызова int-версии функции, в исполняемом коде она будет присутствовать только один раз.

Упрощение листинга

Имейте в виду, что объем оперативной памяти, занимаемой программой, не поменялся из-за того, что мы стали использовать шаблон. Как было три одинаковых функциях, так и осталось. Шаблоны спасают нас только от необходимости вручную писать три раза одно и то же в исходном файле. Однако это делает листинг короче. К тому же, если возникла задача изменения алгоритма, мы изменяем только одно тело функции вместо трех.

Решающий аргумент

Компилятор принимает решение о том, как именно компилировать функцию, основываясь только на типе данных используемого в шаблоне аргумента (или аргументов). Тип данных, возвращаемый функцией, не играет при этом роли. Это немного похоже на то, как компилятор принимает решение, какую из перегруженных функций вызывать.

Просто еще однин трафарет

Мы уже поняли, что шаблон функции — это на самом деле не функция, так как она не приводит к созданию программного кода в памяти. Это просто модель, трафарет для создания множества функций из одной. Это вполне гармонирует с философией ООП. Как и шаблон, класс не представляет собой ничего конкретного, это только база для создания множества похожих объектов.

Шаблоны функций с несколькими аргументами

Взглянем на другой пример шаблона функции. У него три аргумента, два из которых шаблонные, а один — базового типа. Функция предназначена для поиска в массиве заданного числа. Она возвращает индекс найденного значения или −1 в случае его отсутствия в массиве. Аргументами являются указатель на массив, значение, которое нужно искать, а также размер массива. В main() мы определяем четыре различных массива различных типов и четыре значения, которые нужно найти. Тип char в данном примере воспринимается как число. Для каждого массива вызывается шаблонная функция.

Листинг 14.2. Программа TEMPFIND

```
// tempfind.cpp
// Шаблон функции поиска в массиве
#include <iostream>
using namespace std:
```

```
//---------------------------------------------------------
// функция возвращает индекс или -1 при отсутствии //совпадения
template <class atype>
int find(atype* array, atype value, int size)
  {
  for(int j=0; j<size; j++)
    if(array[j]==value)
      return j;
  return -1;
  }
//---------------------------------------------------------
char chrArr[] =   {1, 3, 5, 9, 11, 13}; //массив
char ch = 5;                            //искомое значение
int intArr[] =    {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] =   {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

int main()
  {
  cout << "\n 5 в chrArray: индекс=" << find(chrArr, ch, 6);
  cout << "\n 6 в intArray: индекс=" << find(intArr, in, 6);
  cout << "\n11 в lonArray: индекс=" << find(lonArr, lo, 6);
  cout << "\n 4 в dubArray: индекс=" << find(dubArr, db, 6);
  cout << endl;
  return 0;
  }
```

Здесь, как видите, мы называем шаблонный аргумент именем atype. Оно появляется два раза в аргументах: как тип указателя на массив и как тип искомого значения. Третий аргумент — размер массива — всегда типа int. Это не шаблонный аргумент. Вот результаты работы программы:

```
5 в chrArray: индекс=2
6 в intArray: индекс=-1
11 в lonArray: индекс=4
4 в dubArray: индекс=-1
```

Компилятор генерирует четыре разных варианта функции, по одному на каждый тип. Функция находит число 5 в ячейке массива с индексом 2, не находит число 6 в массиве вообще и т. д.

Аргументы шаблона должны быть согласованы

При вызове шаблонной функции все экземпляры данного аргумента шаблона должны быть строго одного типа. Например, в find() есть массив типа int, соответственно, искомое значение должно быть того же типа. *Нельзя* делать так:

```
int intarray[]={1,3,5,7};   //массив int
float f1 = 5.0;             //значение float
int value = find(intarray, f1, 4);//ах, ох!
```

Компилятору нужно, чтобы все экземпляры atype были одного типа. Он может сгенерировать код функции

```
find(int*, int, int);
```

но не станет заниматься такими глупостями:

```
find(int*, float, int);
```

потому что первый и второй аргумент должны быть одного типа.

Варианты синтаксиса

Некоторым программистам нравится все писать в строчку — и ключевое слово template, и объявление функции:

```
template<class type> int find(atype* array, atype value, int size)
  {
  //набросаем тут тело функции
  }
```

Конечно, добродушный компилятор счастлив и от такого оформления, но структуру синтаксиса лучше соблюдать, а надписи в одну строчку оставить для расписывания заборов. Все-таки должна быть определенная культура программирования.

Различные аргументы одного шаблона

В шаблоне функции можно использовать несколько шаблонных аргументов. Например, вы подумали, что неплохо бы написать шаблон функции find(), но понятия не имеете, к массивам какого размера его потом нужно будет применять. Если массив слишком большой, может понадобиться тип long для обозначения его размера. С другой стороны, как-то не хочется по умолчанию применять long, потому что он может и не пригодиться. Хочется выбрать как тип размера массива, как и тип хранимых в нем данных прямо во время вызова функции. Для этого можно сделать размер массива еще одним аргументом шаблона. Назовем его btype:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
  {
  for(btype j=0; j<size; j++)  //использование btype
    if(array[j]==value)
      return j;
  return static_cast<btype>(-1);
  }
```

Теперь можно использовать значение как типа int, так и long (и даже любого другого пользовательского) в качестве размера массива. Компилятор при своей работе теперь будет ориентироваться не только на разные типы самого массива и искомого числа, но и на разные типы значений его размера.

Обратите внимание, что количество экземпляров шаблонной функции в памяти резко возрастает с ростом числа аргументов шаблона. Уже два разных аргумента в сочетании с шестью базовыми типами приводят к созданию в памяти 36 копий функции. Функции бывают довольно большими, и могут возникнуть проблемы с тем, что программе требуется слишком много памяти. С другой стороны, не всегда реально вызываются шаблонные функции, несмотря на наличие их шаблонов.

А почему не макросы?

Старосветские программисты, пишущие на С, снова удивятся, почему же мы не используем макросы для создания функций, работающих с разными типами данных? Например, функция abs() могла бы быть определена так:

```
#define abs(n) ( (n<0) ? (-n) : (n) )
```

Это возымело бы тот же эффект, что и от шаблона класса в TEMPABS, потому что здесь используется простое замещение одного текста на другой, а значит, можно указывать любые типы данных. Тем не менее, как уже отмечалось ранее, макросы редко используются в С++. С ними возникают некоторые проблемы. Одна из них заключается в том, что макросы не осуществляют проверку типов. Может быть несколько переменных макроса, которые обязаны быть одного типа, но компилятор не проверяет это условие. Еще одна проблема состоит в том, что не указан тип возвращаемого значения, поэтому компилятор, опять же, не может проверить корректность присваивания. В любом случае, макросы ограничены функциями, которые можно выразить с помощью всего одного выражения. Да есть еще и другие проблемы с макросами, которые привели к выходу их из моды при программировании на С++. В целом, лучше избегать их использования.

Что работает, а что нет

Как узнать, можно ли реализовать данную шаблонную функцию с тем или иным типом данных? Например, можно ли использовать функцию find() из программы TEMPFIND для нахождения С-строки (типа char*) в массиве таких строк? Чтобы выяснить этот факт, необходимо проверить операторы, используемые в функции. Если они все способны обработать такой тип данных, то, может быть, что-нибудь и получится. В find(), между тем, мы сравниваем две переменные с помощью оператора ==, а его нельзя использовать с такими строками. Но можно обратиться к библиотечной функции strcmp(). Таким образом, работать с С-строками у нас не получилось. Зато наш шаблон функции работает с классом string, потому что в нем перегружен оператор сравнения (==).

Начните с обычной функции

При написании шаблонных функций, наверное, лучше начать с обычной функции, которая работает с фиксированным типом данных. Можно модифицировать и отлаживать ее сколько душе угодно, совершенно не беспокоясь о синтаксисе шаблонов и разнообразии типов. Затем, когда все доведено до блеска и работает без перебоев, можно ввернуть определение функции в шаблон и проверить, как он будет работать с различными типами.

Шаблоны классов

Шаблонный принцип можно расширить и на классы. В этом случае шаблоны обычно используются, когда класс является хранилищем данных (хороший тому пример мы увидим в следующей главе «Стандартная библиотека шаблонов (STL)»).

Стеки и связные списки, рассмотренные нами в предыдущих главах, — тоже типичные примеры классов-хранилищ данных. До сих пор все приводимые примеры хранилищ могли содержать данные только одного базового типа. Класс stack из программы STAKARAY (глава 7 «Массивы и строки»), например, мог хранить только значения типа int. Вот немного сокращенная версия этого класса:

```
class Stack
  {
  private:
    int st[MAX];          //целочисленный массив
    int top;              //Индекс вершины стека
  public:
    Stack();              //конструктор
    void push(int var);   //int-аргумент
    int pop();            //возвращает значение int
  };
```

Если бы мы захотели хранить значения типа long, пришлось бы написать абсолютно новое определение класса:

```
class LongStack
  {
  private:
    long st[MAX];         //массив long
    int top;              //Индекс вершины стека
  public:
    LongStack();          //конструктор
    void push(long var);  //int-аргумент
    long pop();           //возвращает значение int
  };
```

Подобным же образом нам пришлось бы создавать классы для хранения данных каждого типа, если бы не шаблоны классов, которые и тут приходят к нам на помощь. С их помощью можно написать такую спецификацию класса, которая сохранит все, что попросят, без лишних вопросов. Создадим вариацию на тему STAKARAY для демонстрации возможностей шаблонов классов.

Листинг 14.3. Программа TEMPSTAK

```
// tempstak.cpp
// реализация стека в виде шаблона
#include <iostream>
using namespace std;
const int MAX = 100;
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
  {
  private:
    Type st[MAX];        //стек: массив любого типа
    int top;             //индекс вершины стека
  public:
    Stack()              //конструктор
      { top = -1; }
    void push(Type var) //занести число в стек
      { st[++top] = var; }
```

```
    Type pop()          //вынуть число из стека
      { return st[top--]; }
  };
/////////////////////////////////////////////////////////
int main()
  {
  Stack<float> s1;      //s1 - объект класса Stack<float>

  s1.push(1111.1F);     //занести 3 значения float,
  s1.push(2222.2F);     // вытолкнуть 3 значения float
  s1.push(3333.3F);
  cout << "1: " << s1.pop() << endl;
  cout << "2: " << s1.pop() << endl;
  cout << "3: " << s1.pop() << endl;

  Stack<long> s2;       //s2 - объект класса Stack<long>

  s2.push(123123123L);  //занести 3 long, вытолкнуть 3 long
  s2.push(234234234L);
  s2.push(345345345L);
  cout << "1: " << s2.pop() << endl;
  cout << "2: " << s2.pop() << endl;
  cout << "3: " << s2.pop() << endl;
  return 0;
  }
```

Здесь Stack является шаблонным классом. Идея шаблонных классов во многом сходна с идеей шаблонных функций. Ключевые слова template и class Stack говорят о том, что весь класс будет шаблонным.

```
template <class Type>
class Stack
  {
  //Данные и методы используют шаблонный аргумент Type
  };
```

Шаблонный аргумент Type используется вместо фиксированного типа во всех местах спецификации класса, где есть ссылка на тип массива st. Есть три таких места: определение st, тип аргумента функции push() и тип данных, возвращаемых функцией pop().

Шаблоны классов отличаются от шаблонов функций способом реализации. Для создания шаблонной функции мы вызываем ее с аргументами нужного типа. Классы же реализуются с помощью определения объекта, использующего шаблонный аргумент:

```
Stack<float> s1;
```

Такое выражение создаст объект s1. Это стек, в котором хранятся числа типа float. Компилятор резервирует область памяти для данных этого объекта, используя тип float везде, где появляется аргумент Type в спецификации класса. Резервируется место и под методы класса (если они не были помещены в память другим объектом типа Stack<float>). Конечно же, методы используют только тип float. На рис. 14.2 показано, как шаблоны классов и определения конкретных объектов приводят к занесению этих объектов в память.

Создание объекта типа Stack, хранящего объекты каких-то других типов, как в выражении

```
Stack<long> s2;
```

означает не только резервирование другого объема памяти для данных, но и создание нового набора методов, оперирующих типом long.

Имейте в виду, что имя типа s1 состоит из имени класса Stack и *шаблонного аргумента*: Stack<float>. Это отличает его от других классов, которые могут быть созданы из того же шаблона (Stack<int> или Stack<long>).

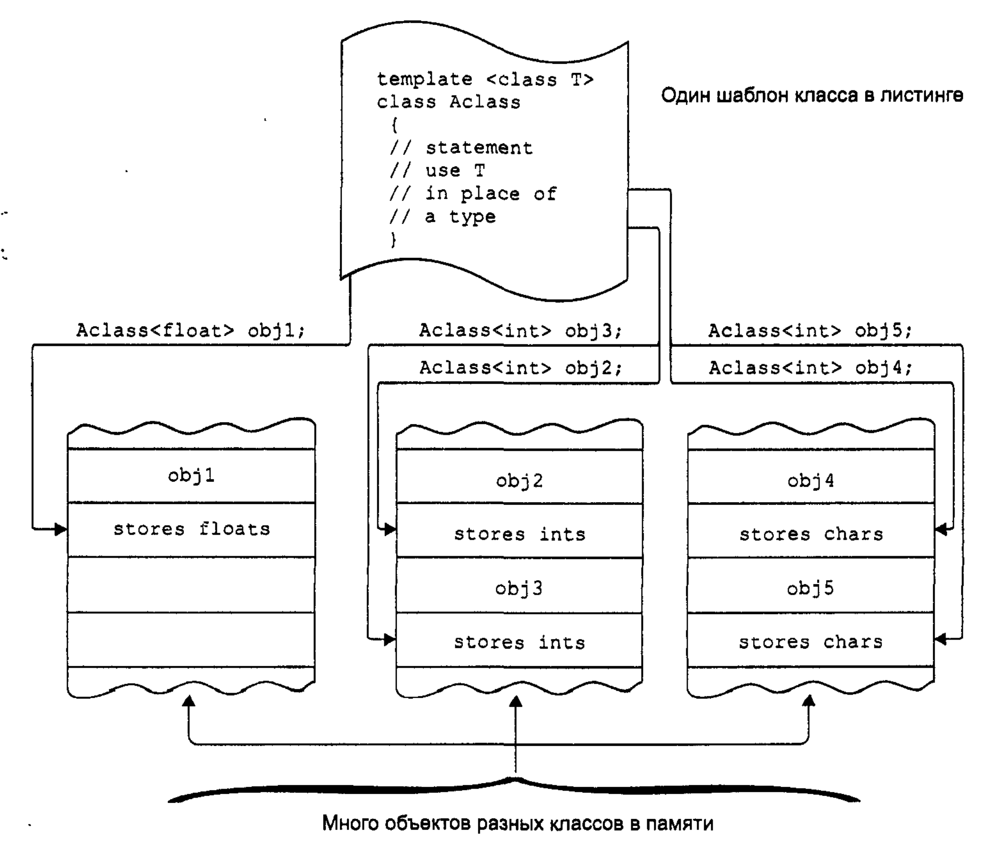


Рис. 14.2. Шаблон класса

В TEMPSTAK мы проверяем стеки s1 и s2, внося в них и выталкивая по три значения и выводя на экран все выталкиваемые значения. Вот результаты работы программы:

```
1: 3333.3//стек float
2: 2222.2
3: 1111.1
1: 345345345//стек long
2: 234234234
3: 123123123
```

В этом примере мы «получаем два класса по цене одного». Для других типов данных можно создать свое множество объектов, и делается это всего одной строкой программы.

Контекстозависимое имя класса

В предыдущем примере все методы шаблона класса были определены внутри класса. Если методы определяются вне спецификации класса, нужно использовать новый синтаксис. Следующая программа показывает, как это можно сделать.

Листинг 14.4. Реализация стека в виде шаблона класса

```
// temstak2.cpp
// реализует стековый класс в виде шаблона
// методы определяются вне класса
#include <iostream>
using namespace std;
const int MAX = 100;
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
  {
  private:
    Type st[MAX];          //стек - массив любого типа
    int top;               //индекс вершины стека
  public:
    Stack();               //конструктор
    void push(Type var);   //занести число в стек
    Type pop();            //взять число из стека
  };
////////////////////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack()       //конструктор
  {
  top = -1;
  }
//--------------------------------------------------------------
template<class Type>
void Stack<Type>::push(Type var) //положить число в стек
  {
  st[++top] = var;
  }
//--------------------------------------------------------------
template<class Type>
Type Stack<Type>::pop()          //взять число из стека
  {
  return st[top--];
  }
//--------------------------------------------------------------

int main()
  {
  Stack<float> s1;         //s1 - объект класса Stack<float>

  s1.push(1111.1F);        //занести 3 float, вытолкнуть 3 float
```

продолжение

Листинг 14.4 (*продолжение*)

```
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;

Stack<long> s2;         //s2 - объект класса Stack<long>

s2.push(123123123L);    //занести 3 long, вытолкнуть 3 long
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}
```

Выражение template<class Type> должно предварять не только определение класса, но и каждый определенный вне класса метод. Вот как выглядит функция push():

```
template<class Type>
void Stack<Type>::push(Type var)
  {
  st[++top] = var;
  }
```

Имя Stack<Type> используется для того, чтобы идентифицировать класс, чьим методом является push().

Для обычного, нешаблонного метода одного имени Stack было бы достаточно:

```
void Stack::push(int var) //Stack() как нешаблонная функция
  {
  st[++top] = var;
  }
```

но для шаблона функции приходится использовать еще и шаблонный аргумент: Stack<Type>.

Итак, имя шаблонного класса выглядит по-разному в зависимости от контекста. Внутри спецификации класса это просто само имя: Stack. Для методов, определенных вне классов, это имя класса и имя шаблонного аргумента: Stack<Type>. Когда вы определяете реальные объекты для хранения конкретного типа данных, именем шаблонного класса будет являться имя класса и имя типа: Stack<float>. Например:

```
class Stack                     //Описатель класса Stack
  { };
void Stack<Type>:.push(Type var) //определение push()
  { }
Stack<float> s1;                //объект типа Stack<float>
```

Нужно быть внимательным, чтобы корректно писать имена, соответствующие контексту. Очень просто можно забыть про <Type> или <float>, которые нужно добавлять к Stack. Компилятор не переносит таких ошибок.

Хотя это и не акцентировалось в предыдущем примере, но нужно всегда крайне аккуратно отслеживать правильность синтаксиса, когда метод возвращает значение своего собственного класса. Предположим, мы определили класс Int, который защищает законные права класса целых чисел (это обсуждалось в упражнении 4 главы 8 «Перегрузка операций»). Если бы вы использовали внешнее определение метода xfunc() данного класса, который возвращает значения типа Int, тогда пришлось бы писать Int<Type> и в качестве типа функции, и в качестве левой части оператора явного задания функции:

```
Int<Type> Int<Type>::xfunc(int arg)
  { }
```

Имя класса, являющееся типом аргумента функции, с другой стороны, вовсе не обязано включать в себя обозначение <Type>.

Создание класса связных списков с помощью шаблонов

Рассмотрим еще один пример, в котором шаблоны помогают в создании классов-хранилищ данных. Для этого мы модифицируем программу LINKLIST из главы 10 «Указатели», которую советуем посмотреть сейчас еще разок. Требуется, чтобы не только класс linklist был переделан в шаблон, но и чтобы структура link, которая реально хранит каждый элемент данных, тоже стала шаблоном.

Листинг 14.5. Программа TEMPLIST

```
// templist.cpp
// Шаблон связных списков
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
template<class TYPE>        //структура link<TYPE>
struct link                 //элемент списка
//( внутри этой структуры определение link означает
// link<TYPE> )
  {
  TYPE data;                //элемент данных
  link* next;               //указатель на следующий элемент
  };
////////////////////////////////////////////////////////////////
template<class TYPE>        //класс linklist<TYPE>
class linklist              //список ссылок
//(внутри этого класса linklist означает linklist<TYPE>)
  {
  private:
    link<TYPE>* first;      //указатель на первую ссылку
  public:
    linklist()              //конструктор без аргументов
      { first = NULL; }     //первой ссылки нет
//примечание: можно вписать еще деструктор: для упрощения //листинга, не приводим его
здесь.
    void additem(TYPE d);   //добавить данные (одна ссылка)
    void display();         //вывести все ссылки
  };
```

продолжение ⇨

Листинг 14.5 (*продолжение*)

```
////////////////////////////////////////////////////////////
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)    //добавление данных
  {
  link<TYPE>* newlink = new link<TYPE>; //создать новую
                                        // ссылку
  newlink->data = d;      //занести туда данные
  newlink->next = first;  //то есть ссылку на следующий элемент
  first = newlink;        //теперь первая ссылка указывает на
                          //данную ссылку
  }
//----------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display() //вывод всех ссылок
  {
  link<TYPE>* current = first; //указатель - на первую ссылку
  while( current != NULL )     //выйти после последней ссылки
    {
    cout << endl << current->data; //вывести на экран
    current = current->next;       //сдвинуться на следующую ссылку
    }
  }
//----------------------------------------------------------
int main()
  {
  linklist<double> ld;    //ld - объект linklist<double>

  ld.additem(151.5);      //добавить три числа double в список ld
  ld.additem(262.6);
  ld.additem(373.7);
  ld.display();           //вывести весь список ld

  linklist<char> lch;     //lch - объект linklist<char>

  lch.additem('a');       //три символа (char) - в список lch
  lch.additem('b');
  lch.additem('c');
  lch.display();          //вывести весь список lch
  cout << endl;
  return 0;
  }
```

В main() мы определяем два связных списка: один — для хранения чисел типа double, другой — для хранения символов (типа char). В каждый из них заносим по три значения с помощью метода additem() и выводим все значения на экран с помощью display(). Результаты работы программы:

```
373.7
262.6
151.5
c
b

a
```

И класс linklist, и структура link пользуются шаблонным аргументом TYPE для подстановки любого типа данных (ну, конечно, не совсем любого: мы позднее

обсудим, какие типы реально могут использоваться). Таким образом, не только linklist, но и link должны быть именно шаблонами, начинающимися со строки

```
template<class TYPE>
```

Вы обратили внимание на то, что не обязательно только классы или функции могут быть превращены в шаблоны? Такое волшебное превращение имеет право на существование для любых элементов программ, использующих изменяющиеся типы данных, как структура link в приведенном примере.

Как и прежде, необходимо следить за именованием класса (и структуры, в этом примере) в разных частях программы. Внутри его собственной спецификации мы используем только имя класса (или структуры): linklist (или link). Во внешних методах мы пишем имя класса, а за ним — имя шаблонного аргумента: linklist<TYPE>. Наконец, когда мы определяем объекты класса, мы уже используем конкретный нужный тип данных, что отражается на записи выражения:

```
linklist<double> ld;      //определяет обьект ld класса
                          //linklist<double>
```

Хранение пользовательских типов

Нам до сих пор удавалось хранить с помощью шаблонных классов только базовые типы данных. В TEMPLIST мы хранили числа типа double и символы типа char в связном списке. Возможно ли хранение пользовательских типов данных с использованием шаблонов? Ответ таков: можно, но осторожно.

Связный список работников

Рассмотрим класс employee из программы EMPLOY (глава 9 «Наследование»). О порожденных классах можно сейчас не беспокоиться. Мы займемся немного другим вопросом. Можно ли хранить объекты этого класса в связном списке из примера TEMPLIST? Как и в случае шаблонных функций, надо определить, может ли шаблонный класс работать с объектами данного класса. Для этого нужно проверять, какие операции он выполняет над этими объектами. Класс linklist использует перегружаемую операцию вставки (<<) для вывода хранящихся в нем объектов:

```
void linklist<TYPE>::display()
  {
  . . .
  cout << endl << current->data; //используется <<
  . . .
  };
```

Никаких проблем с базовыми классами, для которых оператор вставки и так определен, не возникает. Но, к сожалению, в классе employee в программе EMPLOY этот оператор не перегружается. Поэтому придется поработать с классом и внести соответствующие изменения. Для упрощения процедуры получения данных о работниках от пользователя перегрузим-ка мы и оператор извлечения (>>). Данные, полученные в результате извлечения из потока, сохраним во временном объекте emptemp, прежде чем добавлять их в связный список. Приведем листинг программы TEMLIST2.

Листинг 14.6. Программы TEMLIST2

```
// temlist2.cpp
// связный список с использованием шаблона
// список, умеющий работать с классом employee
#include <iostream>
using namespace std;

const int LEN = 80;        //Максимальная длина имен
////////////////////////////////////////////////////////////
class employee             //класс employee
  {
  private:
    char name[LEN];        //имя работника
    unsigned long number;  //номер работника
  public:
    friend istream& operator >> (istream& s, employee& e);
    friend ostream& operator << (ostream& s, employee& e);
  };
//----------------------------------------------------------
istream& operator >> (istream& s, employee& e)
  {
  cout << "\n  Введите фамилию: "; cin >> e.name;
  cout << "  Введите номер: ";     cin >> e.number;
  return s;
  }
//----------------------------------------------------------
ostream& operator << (ostream& s, employee& e)
  {
  cout << "\n  Имя: " << e.name;
  cout << "\n  Номер: " << e.number;
  return s;
  }
////////////////////////////////////////////////////////////
template<class TYPE>       //структура "link<TYPE>"
struct link                //один элемент списка
  {
  TYPE data;               //элемент данных
  link* next;              //указатель на следующую ссылку
  };
////////////////////////////////////////////////////////////
template<class TYPE>       //класс "linklist<TYPE>"
class linklist             //список ссылок
  {
  private:
    link<TYPE>* first;     //указатель на первую ссылку
  public:
    linklist()             //конструктор без аргументов
      { first = NULL; }    //первой ссылки нет
    void additem(TYPE d);  //добавить данные (одну ссылку)
    void display();        //показать все ссылки
  };
//----------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::additem(TYPE d) //добавить данные
  {
  link<TYPE>* newlink=new link<TYPE>;//создать новую ссылку
  newlink->data = d;                 //внести в нее данные
```

```
  newlink->next = first;          //указывающие на следующую ссылку
  first = newlink; //теперь первая ссылка указывает на эту
  }
//----------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display()  //вывод всех ссылок
  {
  link<TYPE>* current = first;  //установить указатель на
                                //первую ссылку
  while( current != NULL )      //выйти после последней ссылки
    {
    cout << endl << current->data; //вывести данные
    current = current->next;       //перейти на следующую ссылку
    }
  }
/////////////////////////////////////////////////////////////
int main()
  {                           //lemp - объект класса
  linklist<employee> lemp;    //"linklist<employee>"
  employee emptemp;           //временное хранилище объектов
  char ans;                   //ответ пользователя ('y' или 'n')

  do
    {
    cin >> emptemp;          //получить данные от пользователя
    lemp.additem(emptemp);   //добавить в emptemp
    cout << "\nПродолжать (y/n)? ";
    cin >> ans;
    } while(ans != 'n');     //при окончании ввода
  lemp.display();            //вывести весь связный список
  cout << endl;
  return 0;
  }
```

В main() мы реализуем связный список lemp. Затем в цикле просим пользователя ввести данные о работниках, после чего добавляем этого работника (то есть соответствующий объект) в список. Когда пользователь заканчивает ввод данных, выводим всю накопленную информацию. Вот один из примеров работы программы:

```
Введите фамилию: Рожанский
Введите номер:1233
Продолжать (y/n)? y
Введите фамилию: Штофф
Введите номер:2344
Продолжать (y/n)? y
Введите фамилию: Ершова
Введите номер:3455
Продолжать (y/n)? n

Имя: Ершова
Номер: 3455
Имя: Штофф
Номер: 2344
Имя: Рожанский
Номер: 1233
```

Обращаем ваше внимание на то, что класс linklist не нужно никак модифицировать, чтобы он хранил объекты пользовательского типа (employee). Вот в этом — вся прелесть шаблонных классов: они работают не только с базовыми типами, но и с типами, определенными пользователями!

Что можно хранить?

Мы уже выяснили, как определить, переменные какого типа могут храниться в шаблонном классе. Для этого нужно понять, какие операции выполняются в его методах. Вот, например, можно ли хранить в связном списке (класс linklist) строки (класс string)? Методы класса используют операторы вставки (<<) и извлечения (>>), которые в отличных взаимоотношениях со строками, так что никаких препон для хранения строк не намечается. Но если в методах есть операции, несовместимые с каким-то типом, тогда значения этого типа хранить в данном классе нельзя.

UML и шаблоны

Шаблоны (или *параметризованные классы*) представляются на диаграммах классов UML в виде некоторых вариаций на тему символов классов. Имена аргументов шаблонов помещены в пунктирный прямоугольник, располагающийся рядом с верхним правым углом символа класса.

На рис. 14.3 показана диаграмма классов для программы TEMPSTAK (см. раздел «Шаблоны классов» данной главы).

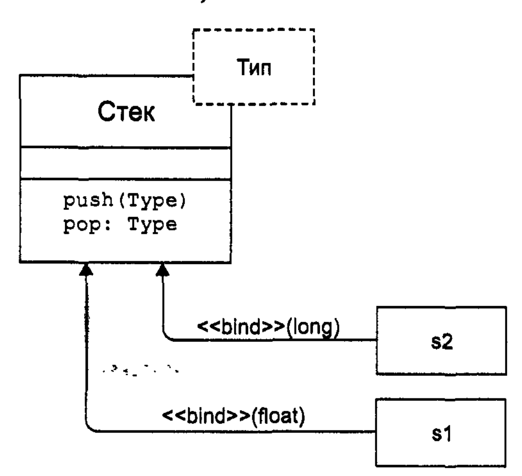


Рис. 14.3. Шаблон на диаграмме классов UML

У шаблона в этой программе только один аргумент — Type. Операции push() и pop() представлены своими типами возвращаемых значений (помните, что тип возвращаемого значения показывается после имени функции и отделяется от него двоеточием). Шаблонный аргумент обычно показывается прямо в обозначении операции, так Type показан на рисунке рядом с push() и pop().

Диаграмма также показывает классы, реализуемые по шаблону: s1, s2.

Кроме шаблонов, на рис. 14.3 показаны две новые концепции UML: *зависимости* и *стереотипы.*

Зависимости в UML

Зависимостью в UML называется отношение между двумя элементами, такое, что изменения в независимом элементе могут привести к изменениям в зависимом. Зависимый элемент, как следует из его названия, зависит от независимого, то есть использует его. Поэтому такие отношения иногда называются отношениями *использования.* В приведенном выше примере шаблонный класс является независимым элементом, а реализуемые классы — зависимыми.

Зависимость изображается на диаграмме пунктирной линией со стрелкой, направленной на независимый элемент. На рис. 14.3 реализованные по шаблону классы s1 и s2 являются зависимыми от шаблонного класса Stack, потому что изменения в последнем приводят или, по крайней мере, могут привести к изменениям в первых.

Вообще, зависимость — это довольно широкое понятие и применяется в UML во многих случаях. Собственно говоря, такие отношения, как ассоциация, обобщение и т. д. являются частными случаями отношения зависимости. Тем не менее они достаточно важны для того, чтобы их изображать на диаграммах в виде отдельных типов отношений.

Одним из характерных примеров ситуации, приводящей к отношению зависимости, является тот случай, когда один класс использует другой в качестве аргумента в своих операциях.

Стереотипы в UML

Стереотип — это способ сообщить дополнительные детали об элементе UML. Он представляется на диаграмме словом в двойных угловых скобках.

Например, пунктирная линия на рис. 14.3 представляет собой зависимость, но ничего не говорит о том, какого она рода-племени. Стереотип <<bind>> сообщает нам о том, что независимый элемент (шаблонный класс) реализует зависимый элемент (s1, например), используя указанные параметры, которые написаны в скобках после стереотипа. То есть в данном случае указывается, что Type будет заменен на float.

UML определяет множество разных стереотипов как элементы языка. Каждый из них применяется к своему элементу: классу, зависимостям и т. д. Можно добавить и что-нибудь свое.

Исключения

Исключения, второй большой вопрос данной главы, позволяют применить систематический, объектно-ориентированный подход к обработке возникающих в классах C++ ошибок. *Исключения* — это ошибки, возникающие во время работы программы. Они могут быть вызваны множеством различных обстоятельств,

таких, как выход за пределы памяти, ошибка открытия файла, попытка инициализировать объект недопустимым значением или использование индекса, выходящего за пределы вектора.

Для чего нужны исключения

Зачем нам понадобился еще один механизм обработки ошибок? Давайте посмотрим, как обрабатывались ошибки в прошлом. Программы на С имели привычку сообщать об ошибке возвращением установленного значения из функции, в которой она произошла. Например, функции работы с дисковыми файлами часто возвращают NULL или 0, чтобы показать, что произошла ошибка. Каждый раз при вызове этих функций происходит проверка возвращаемых значений:

```
if( somefunc() == ERROR_RETURN_VALUE )
  //обработка ошибки или вызов обработчика ошибок
else
  //нормальная работа
if( anotherfunc() == NULL )
  //обработка ошибки или вызов обработчика ошибок
else
  //нормальная работа
if( thirdfunc() == 0 )
  //обработка ошибки или вызов обработчика ошибок
else
  //нормальная работа
```

Одна из проблем данного подхода заключается в том, что каждый вызов такой функции должен проверяться программой. Окружение каждой функции if, добавление выражений обработки ошибок или вызова обработчиков приводит к тому, что код разрастается, как на дрожжах, становится запутанным, читается с трудом.

Проблема еще более усугубляется при использовании классов, так как ошибки могут возникать и при отсутствии явных вызовов функций. Например, представим себе, что какое-либо приложение определяет объекты класса:

```
someClass obj1, obj2, obj3;
```

Как объяснить программе, что произошла ошибка в конструкторе класса? Последний вызывается неявным образом, поэтому никакого возвращаемого значения от него не дождешься.

Еще больше краски сгущаются, когда в приложении используются библиотеки классов. Библиотека классов — это нечто, сделанное посторонними людьми, но используемое вами, программистами, купившими ее, в вашей программе. Поэтому очень сложно отследить ошибки и установить какие-то значения, которые сообщали бы об ошибках, — реализация библиотечных классов всегда остается неизвестной для программиста. Собственно говоря, проблема выявления ошибок, возникающих в глубинах библиотечных функций, — это, возможно, одна из самых глобальных задач, решаемых с помощью исключений. В конце этого раздела мы обязательно вернемся к этому вопросу.

Снова обратимся к исторической памяти программистов, пишущих на С. Они, должно быть, еще помнят те времена, когда в ходу был подход к обработке

ошибок с помощью пары функций setjmp() и longjmp(). Все бы хорошо, но для объектно-ориентированного программирования это не годится, поскольку таким способом невозможно должным образом обрабатывать деструкцию объектов.

Синтаксис исключений

Представим себе приложение, создающее объекты какого-либо класса и работающее с ними. В нормальной ситуации вызовы методов не приводят ни к каким ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает метод. Он информирует приложение о случившемся. При использовании исключений это событие называется генерацией исключительной ситуации. В приложении при этом имеется отдельная секция кода, в которой содержатся операции по обработке ошибок. Этот код называют *обработчиком исключительных ситуаций* или *улавливающим блоком*. В нем отлавливаются исключения, сгенерированные методами. Любой код приложения, использующий объекты класса, заключается в *блок повторных попыток*. Соответственно, ошибки, возникшие в последнем, будут пойманы улавливающим блоком. Код, который не имеет никакого отношения к классу, не включается в блок повторных попыток. На рис. 14.4 показана схема, описанная выше.

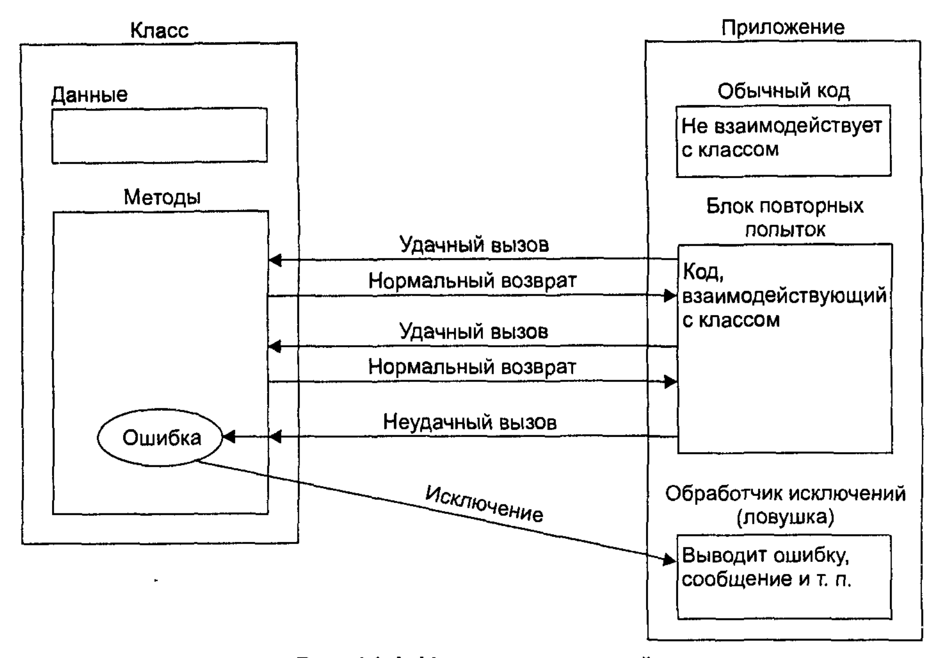


Рис. 14.4. Механизм исключений

Механизм исключений использует три новых слова C++: catch, throw и try. Кроме того, нужно создать новый тип сущности, называемый классом exception. Программа XSYNTAX — это еще не нормальная программа, а только скелет, приводимый для того, чтобы показать синтаксис.

Листинг 14.7. Скелет программы XSYNTAX

```
// xsyntax.cpp
// эта программа не работает!
///////////////////////////////////////////////////////////
class AClass              // просто класс
  {
  public:
  class AnError           // класс exception
    {
    };
  void Func()             //какой-то метод
    {
    if( /* условие ошибки */ )
      throw AnError();    // генерировать исключение
    }
  };
///////////////////////////////////////////////////////////
int main()                // приложение как бы
  {
  try                     // блок повторных попыток
    {
    AClass obj1;          // взаимодействие с объектами AClass
    obj1.Func();          // тут может возникнуть ошибка
    }
  catch(AClass::AnError)  //обработчик ошибок
    {                     // (улавливающий блок)
    }
  return 0;
  }
```

Мы начинаем работу с создания класса AClass, представляющего собой некий класс, в котором могут возникнуть ошибки. Класс исключений, AnError, определен в общедоступной части класса AClass. Методы последнего подвергаются тщательному обследованию на предмет выявления ошибок. Если таковые обнаруживаются, мы с помощью ключевого слова throw и конструктора класса исключений производим генерацию исключительной ситуации:

```
throw AnError();          //'throw' и конструктор класса AnError
```

В main() мы заключаем все выражения, каким-либо образом относящиеся к AClass, в блок повторных попыток. Если какие-то из них приводят к тому, что возникают подозрения на наличие ошибки в методе AClass, то генерируется исключение, и управление программой переходит к улавливающему блоку, который следует сразу же за блоком повторных попыток.

Простой пример исключения

Давайте, наконец, посмотрим на какую-нибудь реально работающую программу, использующую идею исключений. Пример, который вы сейчас увидите, является развитием программы STAKARAY из главы 7; в ней, если помните, создавалась стековая структура данных для хранения целочисленных значений. К сожалению, тот более ранний пример не обнаруживал двух важных ошибок. А именно: приложение могло пытаться извлечь или поместить в стек слишком много объ-

ектов, что привело бы либо к получению неправильных данных, либо к превышению размера массива. В программе XSTAK мы используем механизм исключений, чтобы обрабатывать эти ситуации.

Листинг 14.8. Программа XSTAK

```
// xstak.cpp
// Демонстрация механизма исключений
#include <iostream>
using namespace std;
const int MAX = 3;          //в стеке максимум 3 целых числа
///////////////////////////////////////////////////////////
class Stack
  {
  private:
    int st[MAX];            //стек: целочисленный массив
    int top;                //индекс вершины стека
  public:
    class Range             //класс исключений для Stack
      {                     //внимание: тело класса пусто
      };
//---------------------------------------------------------
    Stack()                 //конструктор
      { top = -1; }
//---------------------------------------------------------
    void push(int var)
      {
      if(top >= MAX-1)      //если стек заполнен,
        throw Range();      //генерировать исключение
      st[++top] = var;      //внести число в стек
      }
//---------------------------------------------------------
    int pop()
      {
      if(top < 0)           //если стек пуст,
        throw Range();      //исключение
      return st[top--];     //взять число из стека
      }
  };
///////////////////////////////////////////////////////////
int main()
  {
  Stack s1;

  try
    {
    s1.push(11);
    s1.push(22);
    s1.push(33);
//    s1.push(44);          //Опаньки! Стек заполнен
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;
    cout << "4: " << s1.pop() << endl;
                            //Опаньки! Стек пуст
    }
```

продолжение

Листинг 14.8 (*продолжение*)

```
catch(Stack::Range)      //обработчик
   {
   cout << "Исключение: Стек переполнен или пуст"<<endl;
   }
cout << "Приехали сюда после захвата исключения (или нормального выхода" << endl;
return 0;
}
```

Имейте в виду, мы специально сделали стек таким маленьким, чтобы проще было привести пример возникновения исключительной ситуации, внеся в него слишком много элементов.

Рассмотрим части этой программы, относящиеся к исключениям. Их четыре. В спецификации класса имеется класс исключений. Также там имеются выражения, генерирующие исключения. В main() есть часть кода, которая может привести к исключительным ситуациям, — это блок повторных попыток. Есть код, в котором эти ситуации обрабатываются, — улавливающий блок.

Описание класса исключений

В нашей программе класс исключений описывается внутри класса Stack:

```
class Range
  {     //Тело класса пусто!
  };
```

Как видите, тело класса пусто, а потому его объекты не обладают ни данными, ни методами. Все, что нам на самом деле нужно в этой простенькой программе, это имя класса Range. Оно используется для связывания выражения генерации исключения с улавливающим блоком (классу не обязательно быть пустым, как мы увидим позднее).

Генерация исключения

В классе Stack исключение возникает, когда приложение пытается извлечь значение из пустого стека или занести значение в заполненный до отказа стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом класса Stack, методы этого класса проверяют указанные условия с использованием if и генерируют исключение, если эти условия выполняются. В программе XSTAK исключение генерируется в двух местах, оба раза с помощью выражения

```
throw Range();
```

Второе слово этого незамысловатого предложения неявно запускает конструктор класса Range для создания, соответственно, объекта этого класса. Что касается первого слова, то оно передает управление программой обработчику ошибок (мы вскоре его увидим).

Блок повторных попыток (Try-блок)

Все выражения в main(), в которых могут произойти ошибки, то есть выражения, манипулирующие данными объектов класса Stack, заключены в фигурные скобки, перед которыми стоит ключевое слово try:

```
try
  {
  //код, работающий с объектами, в которых могут
  //произойти ошибки
  }
```

Это просто часть обычного кода программы. Такое могло бы понадобиться вне зависимости от того, используется механизм исключений или нет. В блок повторных попыток включаются только выражения, которым по долгу службы приходится иметь дело с классом Stack. Более того, в программе может быть несколько таких try-блоков, чтобы к этому классу был доступ из разных мест.

Обработчик ошибок (улавливающий блок)

Код, в котором содержатся операции по обработке ошибок, заключается в фигурные скобки и начинается с волшебного слова catch. Имя класса исключений должно включать в себя наименование класса, в котором он живет. В данном примере это Stack::Range.

```
catch(Stack::Range)
  {
  //код-обработчик ошибок
  }
```

Такая конструкция называется обработчиком прерываний. Она должна следовать непосредственно за блоком повторных попыток. Что касается приведенного примера, то в нем обработчик ошибок занимается только выводом на экран сообщения об ошибках, чтобы пользователь знал, почему его программа перестала работать. Управление программой «проваливается» на дно обработчика ошибок, так что программа может продолжать работу прямо с этого места. Обработчик, правда, может передать управление совсем в другое место, а еще чаще он поступает совсем просто: завершает работу программы.

Последовательность событий

Давайте подведем итоги и покажем последовательность действий программы при возникновении ошибки.

1. Код нормально выполняется вне блока повторных попыток.
2. Управление переходит в блок повторных попыток.
3. Какое-то выражение в этом блоке приводит к возникновению ошибки в методе.
4. Метод генерирует исключение.
5. Управление переходит к обработчику ошибок (улавливающему блоку), следующему сразу за блоком повторных попыток.

Вот и все. Обратите внимание, каким прозрачным и четким получился конечный алгоритм. В любом выражении блока повторных попыток может произойти ошибка, но у нас все под контролем, не нужно беспокоиться о получении возвращаемого значения для каждого из них, потому что это делается автоматиче-

ски. В данном конкретном примере мы сознательно написали два ошибочных выражения. Первое:

```
s1.push(44);       //в стек занесено слишком много элементов
```

приводит к исключительной ситуации (если в примере убрать комментарии перед ним), а второе:

```
cout << "4: " << s1.pop() << endl; //извлечение из пустого
                                   //стека
```

приводит к исключительной ситуации, если закомментировать первое выражение. Попробуйте сделать и так, и так. В обоих случаях будет выведено одно и то же сообщение:

```
Исключение: Стек переполнен или пуст
```

Многократные исключения

Можно спроектировать класс таким образом, чтобы он генерировал столько исключений, сколько нужно. Чтобы показать, как это делается, мы изменим программу XSTAK таким образом, чтобы она выдавала отдельные исключения при попытке переполнения стека и попытке извлечения из пустого стека.

Листинг 14.9. Программа XSTAK2

```
// xstak2.cpp
// Демонстрация обработчика двух исключений
#include <iostream>
using namespace std;
const int MAX = 3; //в стеке может быть до трех целых чисел
///////////////////////////////////////////////////////////
class Stack
  {
  private:
    int st[MAX];          //стек: массив целых чисел
    int top;              //индекс верхушки стека
  public:
    class Full { };       //класс исключения
    class Empty { };      //класс исключения
//---------------------------------------------------------
    Stack()               //конструктор
      { top = -1; }
//---------------------------------------------------------
    void push(int var)    //занести число в стек
      {
      if(top >= MAX-1)    //если стек полон,
        throw Full();     //генерировать исключение Full
      st[++top] = var;
      }
//---------------------------------------------------------
    int pop()             //взять число из стека
      {
      if(top < 0)         //если стек пуст,
        throw Empty();    //генерировать исключение Empty
      return st[top--];
```

```
      }
  };
////////////////////////////////////////////////////////////////
int main()
  {
  Stack s1;

  try
    {
    s1.push(11);
    s1.push(22);
    s1.push(33);
//   s1.push(44);         //Опаньки: стек уже полон
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;
    cout << "4: " << s1.pop() << endl;
                          //Опаньки: стек пуст
    }
  catch(Stack::Full)
    {
    cout << "Ошибка: переполнение стека" << endl;
    }
  catch(Stack::Empty)
    {
    cout << "Ошибка: стек пуст" << endl;
    }
  return 0;
  }
```

В этой программе были описаны два класса исключений:

```
class Full { };
class Empty { };
```

Выражение

```
throw Full();
```

исполняется, если программа вызывает push() при уже заполненном стеке, а

```
throw Empty();
```

исполняется, если программа вызывает pop() при пустом стеке.

Для каждого исключения используется отдельный отлавливающий блок:

```
try
  {
  //код, работающий с объектами класса Stack
  }
catch
  {
  //код обработки исключения Full
  }
catch
  {
  //код обработки исключения Empty
  }
```

Все отлавливающие блоки, используемые с конкретными блоками повторных попыток, должны следовать в коде непосредственно за ними. В этом случае каждое исключение приведет к выводу своего сообщения об ошибке: «Стек заполнен» или «Стек пуст». Даешь каждому отлавливающему блоку по своему исключению! Таков девиз приведенной выше программы, да так, собственно, зачастую и делается в программах. Группа отлавливающих блоков (*цепь ловушек*) работает примерно как выражение switch в том смысле, что исполняется только один соответствующий блок кода. После обработки исключения управление программой переходит к тому месту программы, которое следует за всеми блоками-ловушками. (В отличие от switch, не нужно заканчивать каждый отлавливающий блок break. В какой-то мере эти блоки напоминают функции.)

Исключения и класс Distance

В качестве еще одного примера технологии исключений рассмотрим ее применение к уже родному классу Distance. Объекты этого класса по-прежнему состоят из целочисленных значений футов и значений дюймов типа float. При этом число дюймов не должно превышать 12,0. Проблема этого класса в предыдущих программах заключалась в том, что невозможно было защититься от ввода пользователем неправильного значения дюймов. Это могло приводить к некорректности данных, получаемых в результате выполнения арифметических операций, так как в этих операциях предполагалось, что значения дюймов все-таки меньше 12,0. Недопустимые значения тоже могли быть выведены, а пользователь мог впасть в легкий ступор наедине с форматом 7'-15".

Перепишем теперь класс Distance таким образом, чтобы можно было использовать механизм исключений для обработки ошибок.

Листинг 14.10. Программа XDIST

```
// xdist.cpp
// Исключения и класс Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance            //класс английских мер расстояний
  {
  private:
    int feet;
    float inches;
  public:
    class InchesEx { };   //класс исключений
//--------------------------------------------------------------
    Distance()            //конструктор (без аргументов)
      { feet = 0; inches = 0.0; }
//--------------------------------------------------------------
    Distance(int ft, float in) //конструктор (2 арг-та)
      {
      if(in >= 12.0)          //если дюймы указаны неверно,
        throw InchesEx();     //генерировать исключение
      feet = ft;
      inches = in;
      }
```

```
//-----------------------------------------------------------
   void getdist()        //получить длину от пользователя
     {
     cout << "\nEnter feet: ";  cin >> feet;
     cout << "Enter inches: ";  cin >> inches;
     if(inches >= 12.0)  //если дюймы неправильные,
       throw InchesEx(); //генерировать исключение
     }
//-----------------------------------------------------------
   void showdist()          //вывод расстояний
     { cout << feet << "\'-" << inches << '\"'; }
 };
/////////////////////////////////////////////////////////////
int main()
  {
  try
    {
    Distance dist1(17, 3.5); //конструктор (2 аргумента)
    Distance dist2;          //конструктор (без аргументов)
    dist2.getdist();         //получить расстояние
                             //вывести расстояния
    cout << "\ndist1 = ";  dist1.showdist();
    cout << "\ndist2 = ";  dist2.showdist();
    }
  catch(Distance::InchesEx)  //поймать исключения
    {
    cout << "\nОшибка инициализации: "
      "значение дюймов превышает предельно допустимое.";
    }
  cout << endl;
  return 0;
  }
```

Итак, разберемся с этой программой. Здесь мы создали класс исключений, называемый InchesEx, для класса Distance. При любой попытке пользователя ввести неправильное значение генерируется исключение. Так случается в двух местах: в конструкторе аргументов, где программист может неверно задать инициализирующие значения, а также в функции getdist(), где пользователь может ввести ошибочные данные в ответ на просьбу ввести дюймы. Можно проверять таким способом, например, случайный ввод отрицательных значений, да и другие ошибки ввода.

В main() все взаимодействия с объектами Distance заключены в блок повторных попыток, а блок-ловушка выводит сообщение об ошибках.

В серьезно разрабатываемых программах, конечно, можно обрабатывать ошибки пользователя (как бы противопоставляя их ошибкам программиста) по-другому. Более приемлемо, с точки зрения пользователя, было бы, например, не выходить из программы и не просто выводить строку сообщения об ошибке, а возвращаться на начало блока повторных попыток. Собственного говоря, это блок так и называется из-за того, что запуск программы после сбоя в общем случае может производиться с него. Так вот, если вернуться к началу этого блока повторных попыток, то пользователю тем самым может быть дан шанс ввести правильное значение дюймов.

Исключения с аргументами

А что, если приложению нужно больше информации о том, что привело к возникновению исключительно ситуации? Например, в программе XDIST может понадобиться выяснить, какие именно значения дюймов были введены. Далее, если, например, одно и то же исключение генерируется разными методами класса, то бывает важно выяснить, кто же виновник того, что произошла ошибка. Может быть, существует какой-то способ передать такую информацию от метода, где было сгенерировано исключение, к приложению, которое его поймало?

На этот вопрос можно ответить, если вспомнить, что генерация исключения включает в себя не только передачу управления обработчику, но и создание объекта класса исключений с помощью вызова его конструктора. Например, в XDIST мы создаем объект типа InchesEx при генерации исключения с помощью выражения

```
throw InchesEx();
```

Если мы добавляем компонентные данные к классу исключений, мы можем их инициализировать при создании объекта. Обработчик исключений затем может извлечь данные из этого объекта при поимке исключения. С таким же успехом можно сделать какую-нибудь надпись на футбольном мяче и перекинуть его приятелю через забор. Немного модифицируем программу XDIST, чтобы продемонстрировать вышесказанное.

Листинг 14.11. Программа XDIST2

```
// xdist2.cpp
// исключения с аргументами
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance              //класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
//-----------------------------------------------------------------
    class InchesEx          //класс исключений
       {
       public:
         string origin;     //для имени функции
         float iValue;      //для хранения ошибочного
                            //значения

         InchesEx(string or, float in) //конструктор с
                                       //двумя аргументами
           {
           origin = or;                //сохраненная строка
                                       //с именем виновника ошибки
           iValue = in;                //сохраненное неправильно
                                       //значение дюймов
           }
       };                              //конец класса исключений
```

```
//-----------------------------------------------------------
   Distance()                    //конструктор (без аргументов)
     { feet = 0; inches = 0.0; }
//-----------------------------------------------------------
   Distance(int ft, float in) //конструктор (2 аргумента)
     {
     if(in >= 12.0)
       throw InchesEx("Конструктор с двумя аргументами", in);
     feet = ft;
     inches = in;
     }
//-----------------------------------------------------------
   void getdist()            //получить данные от пользователя
     {
     cout << "\nВведите футы: ";  cin >> feet;
     cout << "Введите дюймы: ";  cin >> inches;
     if(inches >= 12.0)
       throw InchesEx("функция getdist()", inches);
     }
//-----------------------------------------------------------
   void showdist()            //вывести расстояние
     { cout << feet << "\'-" << inches << '\"'; }
  };
/////////////////////////////////////////////////////////////
int main()
  {
  try
    {
    Distance dist1(17, 3.5); //конструктор с двумя
                             //аргументами
    Distance dist2;          //конструктор без аргументов
    dist2.getdist();         //получить значение
                             //вывести расстояния
    cout << "\ndist1 = ";  dist1.showdist();
    cout << "\ndist2 = ";  dist2.showdist();
    }
  catch(Distance::InchesEx ix)  //обработчик ошибок
    {
    cout << "\nОшибка инициализации. Виновник: " << ix.origin
       << ".\n  Введенное значение дюймов " << ix.iValue
       << " слишком большое.";
    }
  cout << endl;
  return 0;
  }
```

При передаче данных в обработчик ошибок производятся три действия: задание методов и конструктора класса исключений, инициализация этого конструктора при генерации исключения, обращение к данным объекта при поимке исключения. Давайте разберемся во всем по порядку.

Спецификация данных в классе исключений

Данные в классе исключений лучше всего сделать общедоступными, чтобы обработчик имел прямой доступ к ним. Вот так выглядит спецификация класса исключений InchesEx в программе XDIST2:

```
class InchesEx                  //класс исключений
   {
   public:
     string origin;             //для имени функции
     float iValue;              //для хранения ошибочного
                                //значения
InchesEx(string or, float in)   //конструктор с
                                //двумя аргументами
     {
      origin = or;              //сохраненная строка
                                //с именем виновника ошибки
iValue = in;                    //сохраненное неправильное
                                //значение дюймов
      }
  };                            //конец класса исключений
```

В этом классе имеются общедоступные переменные для объекта класса string, в котором будут содержаться сведения об имени вызываемого метода, а также переменные типа float для хранения ошибочных значений дюймов.

Инициализация объекта класса исключений

Как же инициализировать данные при генерации исключительной ситуации? В конструкторе с двумя аргументами класса Stack мы пишем:

```
throw InchesEx("Конструктор с двумя аргументами", in);
```

А в методе getdist() для класса Stack пишем:

```
throw InchesEx("функция getdist()", inches);
```

При генерации исключения обработчик выведет соответствующую строку и значение количества дюймов. Из строки мы узнаем, в каком методе произошла ошибка, а inches сообщит, какое именно значение ошибочно ввел пользователь. Такой способ обработки исключений позволит программисту или пользователю более точно определить причину сбоя в программе.

Извлечение данных из объекта класса исключений

Как нам добыть данные при поимке исключения? Проще всего сделать их общедоступной частью класса исключений, как мы и сделали в нашем примере. В блоке-ловушке можно объявить ix как имя объекта класса исключений, за которым мы охотимся с нашим большим исключительным ружьем. Используя это имя, можно запросто обращаться к данным объекта, записывая их через точку:

```
catch(Distance::inchesEx ix)
  {
  //прямой доступ к 'ix.origin' и 'ix.value'
  }
```

Можно совершенно спокойно выводить на экран значения ix.origin и ix.value. Приведем пример взаимодействия пользователя с программой XDIST2. Пользователь ввел, как обычно, некорректное значение дюймов, оно превышает значение 12,0.

```
Введите футы: 7
Введите дюймы: 13.5
```

```
Ошибка инициализации. Виновник: функция getdixt()
Введенное значение дюймов 13.5 слишком большое.
```

Если же нерадивому программисту вдруг захочется поменять определение dist1 в main() на

```
Distance dist1(17, 22.25);
```

сгенерируется исключение, в результате чего на экране появится надпись следующего содержания:

```
Ошибка инициализации. Виновник: конструктор с двумя аргументами.
Введенное значение дюймов 22.25 слишком большое.
```

Конечно, с аргументами исключений можно делать все, что угодно, но типичное их использование — это получение дополнительной информации, помогающей диагностировать ошибки.

Класс bad_alloc

В стандарт C++ включено несколько встроенных классов исключений. Наверное, чаще всего в работе используется класс bad_alloc, который генерирует исключение, если возникает ошибка резервирования памяти при использовании new. В более ранних версиях C++ это исключение называлось xalloc. На момент публикации этой книги прежний подход все еще использовался в Microsoft Visual C++. Если правильно установить блок-ловушку и блок повторных попыток, то bad_alloc можно использовать совершенно без проблем. В следующем коротеньком примере BADALLOC показано, как это делается.

Листинг 14.12. Программа BADALLOC

```
// badalloc.cpp
// Демонстрация исключения bad_alloc
#include <iostream>
using namespace std;

int main()
  {
  const unsigned long SIZE = 10000; //объем памяти
  char* ptr;                 //указатель на адрес в памяти

  try
    {
    ptr = new char[SIZE]; //разместить в памяти SIZE байт
    }
  catch(bad_alloc)         //обработчик исключений
    {
    cout << "\n Исключение bad_alloc: невозможно разместить данные в памяти.\n";
    return(1);
    }
  delete[] ptr;            //освободить память
  cout << "\nПамять используется без сбоев.\n";
  return 0;
  }
```

Все выражения, каким-либо образом связанные с new, поместите в блок повторных попыток. Блок-ловушка, который ставится сразу за этим блоком, обработает ошибку, то есть выведет сообщение и закроет программу.

Размышления об исключениях

Мы показали только простейшие примеры использования механизма исключений. Углубляться в данную тему нам не позволяет объем книги, но хочется высказать напоследок несколько соображений относительно этой технологии.

Вложенные функции

Выражение, приводящее к исключительной ситуации, не обязательно располагается непосредственно в блоке повторных попыток. Оно может быть и в функции, вызываемой выражением из этого блока (или функцией, вызванной функцией, вызванной функцией из блока повторных попыток, и т. д., понятное дело). Поэтому следует сделать блок повторных попыток верхним уровнем программы. О функциях, находящихся на более низких уровнях, можно заботиться не столь усердно, поскольку в любом случае они вызываются с верхнего уровня, то есть из блока повторных попыток (тем не менее функциям среднего уровня иногда бывает полезно добавлять данные о себе в исключения или передавать их вниз по иерархии).

Исключения и библиотеки классов

Важной проблемой, которую решает механизм исключений, является выявление ошибок в библиотеках классов. В библиотечной функции легко может возникнуть ошибка, но обычно совершенно неясно, где именно и что с ней делать. Кроме того, надо учитывать, что алгоритмы для библиотеки были написаны какими-то другими людьми, в другое время, то есть довольно сильно могут отличаться по концепции от программы, которую вы собираетесь писать. Что обязана в любом случае библиотечная функция сделать, так это сообщить о возникшей ошибке в вызвавшую ее программу. Проще говоря, библиотечная функция должна сказать: «У меня произошла ошибка. Я не знаю, что с ней делать, но она уже произошла» и вызывающая программа будет обрабатывать эту ошибку по своему усмотрению.

Механизм исключений включает в себя такую возможность, поскольку исключения передаются наверх по иерархической лестнице вложенных функций до тех пор, пока не дойдут до блока-ловушки. Выражение выдачи исключения может входить в библиотеку, но ловушка может быть только в вызывающей программе, которая одна знает, как лучше обработать ошибку.

Если вы взялись за написание библиотеки классов, позаботьтесь о том, чтобы исключения выдавались в вызывающую программу из всех функций, в которых может случиться что-то неординарное. Если же вы пишете программу, использующую библиотеку, то нужно предусмотреть блок повторных попыток и блок-ловушку для любых исключений, которые может выдавать библиотека.

Исключения — не панацея

Механизм исключений невозможно применять ко всем типам ошибок. Есть некий потолок, касающийся размеров программы и времени на обработку ошибок. Например, все-таки, наверное, нецелесообразно использовать исключения для работы с ошибками ввода (например, ввода символов вместо цифр), такие коллизии гораздо проще ликвидировать традиционными способами. Да-да, просто в цикле проверять, что вводит пользователь, и принимать соответствующие решения. В случае необходимости — просить пользователя ввести данные заново.

Автоматически вызываемые деструкторы

Технология обработки исключений удивительно умна и практична. При генерации исключения деструктор вызывается автоматически для того объекта, который был создан в коде как раз перед тем местом, где возникла ошибка. Это действительно необходимо, поскольку приложение не знает, какое выражение стало причиной исключения, а если программа хочет после обработки ошибки продолжать свое выполнение, то ей придется (по меньшей мере) пройти заново весь блок повторных попыток. Механизм исключений гарантирует, что код в try-блоке будет «перезагружен».

Обработка исключений

После успешной ловли исключений вам, как истинному охотнику, вероятно, захочется прикончить всю программу, чтобы никаких исключений больше тут не водилось вообще. Механизм исключений позволяет выявить виновника произошедшей ошибки, представить его имя пользователю, а перед завершением программой произвести все необходимые ритуалы очистки памяти, завершения стартовавших процессов и т. п. Лучшее средство для очистки совести компьютера перед закрытием приложения — это деструктор. Как раз с его помощью можно вывести даже самые невыводимые пятна в памяти компьютера, во всяком случае, той ее части, которая использовалась объектами.

В других ситуациях у вас может не появиться намерения завершить программу. Может быть, ошибку можно исправить «на лету» или попросить пользователя ввести данные корректно и исчерпать тем самым инцидент. В таком случае, блоки повторных попыток и блоки-ловушки замыкаются в цикл так, чтобы после работы обработчика исключений управление в программе вернулось бы к началу блока повторных попыток.

Если не нашлось обработчика для сгенерировавшегося исключения, операционная система бесцеремонно закрывает программу.

Резюме

Шаблоны позволяют создавать семейства функций или семейства классов, выполняющих одни и те же операции с разными типами данных. Если вы вдруг ловите себя на том, что пишете функции, которые делают одно и то же, но с разными типами данных, значит, пришла пора написать шаблон функции. Если вы

ловите себя на том, что пишете уже в который раз спецификацию класса, который делает то же, что и предыдущие, но с другими типами, следует подумать о написании шаблона класса. Этим вы сохраните свое драгоценное время, а в результате получится ясный, понятный и компактный код, причем алгоритмы можно будет менять очень легко и сразу для работы со всеми используемыми типами данных.

Исключения – это механизм обработки ошибок, возникающих при некорректной работе программ, написанных на C++, с использованием систематического, объектно-ориентированного подхода. Исключительные ситуации обычно возникают вследствие ошибочных выражений в блоках повторных попыток (try-блоках), которые работают с объектами некоторого класса. Метод класса обнаруживает ошибку и «генерируют исключение», которое ловится программой с помощью класса, следующего в коде обработчика исключений сразу за блоком повторных попыток.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Шаблоны позволяют удобным способом создавать семейства:
   а) переменных;
   б) функций;
   в) классов;
   г) программ.
2. Шаблонный аргумент всегда начинается с ключевого слова __________.
3. Истинно ли утверждение о том, что шаблоны автоматически создают разные версии класса в зависимости от введенных пользователем данных?
4. Напишите шаблон функции, всегда возвращающей свой аргумент, умноженный на два.
5. Шаблонный класс:
   а) создается для того, чтобы храниться в разных контейнерах;
   б) работает с разными типами данных;
   в) генерирует идентичные объекты;
   г) генерирует классы с различным числом методов.
6. Истинно ли утверждение о том, что шаблон может иметь несколько аргументов?
7. Создание реальной функции из шаблона называется ______________ функции.
8. Реальный код шаблонной функции генерируется при:
   а) объявлении функции в исходном коде;
   б) определении функции в исходном коде;

в) вызове функции в исходном коде;

г) запуске функции во время работы программы.

9. Ключевой для шаблонов является концепция, согласно которой ________ заменяется на имя, которое подставляется вместо ________________.
10. Шаблоны часто используются с классами, которые ________ ________.
11. Исключение в большинстве случаев возникает из-за:

а) программиста, написавшего исходный код приложения;

б) создателя класса, написавшего его методы;

в) ошибки выполнения;

г) сбоя в операционной системе.

12. При работе с механизмом исключений в C++ используются следующие ключевые слова: ____________, ____________, ________________.
13. Напишите выражение, генерирующее исключение, используя класс BoundsError (тело класса пусто).
14. Истинно ли утверждение о том, что выражения, котрые могут создать исключительную ситуацию, должны быть частью блока-ловушки?
15. Исключения передаются:

а) из блока-ловушки в блок повторных попыток;

б) из выражения, создавшего исключительную ситуацию, в блок повторных попыток;

в) из точки, где возникла ошибка, в блок-ловушку;

г) из выражения, в котором возникла ошибка, в блок-ловушку.

16. Напишите спецификацию класса исключений, хранящего номер ошибки и ее название. Включите в класс конструктор.
17. Истинно ли утверждение о том, что выражение, генерирующее исключение, не должно быть расположено в блоке повторных попыток?
18. Для следующих ошибок обычно генерируется исключение:

а) чрезмерное количество данных грозит переполнить массив;

б) пользователь нажал Ctrl+C для закрытия программы;

в) скачок напряжения в сети привел к перезагрузке системы;

г) new не может зарезервировать необходимый объем памяти.

19. Дополнительная информация, передающаяся при генерации исключения, может быть помещена в:

а) ключевое слово throw;

б) функцию, вызвавшую ошибку;

в) блок-ловушку;

г) объект класса исключений.

20. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?

21. Говоря о зависимостях, мы под шаблонным классом подразумеваем ________________ элемент программы, а под реализуемым — ______________.
22. Шаблонный класс показывается на диаграмме UML в виде:
    а) обычного класса с каким-то аппендиксом;
    б) пунктирной линии;
    в) прямоугольника с пунктирным контуром;
    г) нет правильного ответа.
23. Истинно ли утверждение о том, что зависимость — это вид ассоциации?
24. Стереотип дает ________________________ об элементе UML.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Напишите шаблон функции, возвращающей среднее арифметическое всех элементов массива. Аргументами функции должны быть имя и размер массива (типа int). В main() проверьте работу функции с массивами типа int, long, double и char.

*2. Очередь — это тип хранилища данных. Она напоминает по своей сути стек, только вместо правила LIFO (последний вошел — первый вышел) использует правило FIFO (первый вошел — первый вышел). В общем, как очередь у окошка в банке. Если вы заносите в массив числа 1, 2, 3, обратно вы их получите в той же последовательности.

   Стеку требуется для работы только один индекс массива (top, вершина массива. См. программу STAKARAY из главы 7). Очереди же требуется два индекса: индекс хвоста, куда добавляются новые элементы, и индекс головы очереди, откуда исчезают старые. Хвост постепенно сдвигается, как и голова. Если хвост или голова достигает конца массива, он переставляется на начало.

   Напишите шаблон класса для работы с очередью. Предположим, что программист не будет совершать ошибок при написании ее модели. Например, вместимость очереди не будет превышена, а из пустой очереди не будет производиться попыток удаления данных. Определите несколько очередей разных типов и поработайте с их данными.

*3. Добавьте механизм обработки исключений в программу из упражнения 2. Рассмотрите два исключения: при превышении размера очереди и при попытке удаления данных из пустой очереди. Это можно сделать, добавив новый элемент данных в класс очереди — счетчик текущего числа элементов в очереди. Инкрементируйте счетчик при добавлении нового элемента, декрементируйте, соответственно, при удалении элемента из головы очереди. Генерируйте исключение, если счетчик превысил размер массива или если он стал меньше 0.

Можно попробовать сделать main() интерактивной, чтобы пользователь мог вводить и извлекать данные. Так будет проще проверить работу написанных функций. С помощью механизма исключений программа должна обеспечивать повторную попытку ввода данных пользователем без нарушения целостности содержимого очереди.

4. Создайте функцию swaps(), обменивающую значения двух аргументов, посылаемых ей. (Возможно, вы догадаетесь передавать эти аргументы по ссылке.) Сделайте из функции шаблон, чтобы она могла использоваться с любыми числовыми типами данных (int, char, float и т. д.). Напишите main() для тестирования функции.

5. Создайте функцию amax(), возвращающую значение максимального элемента массива. Аргументами функции должны быть адрес и размер массива. Сделайте из функции шаблон, чтобы она могла работать с массивом любого числового типа. Напишите секцию main(), в которой проверьте работу функции с разными типами массивов.

6. Начните работу с класса safearray из программы ARROVER3 (глава 8). Сделайте класс шаблоном, чтобы массив мог хранить любые типы данных. В секции main() создайте, по крайней мере, два массива разных типов.

7. За основу возьмите класс frac и калькулятор с четырьмя функциями из упражнения 7 главы 8. Сделайте этот класс шаблоном, чтобы его можно было реализовывать с использованием различных типов данных в качестве делимого и делителя. Конечно, это должны быть целочисленные типы, что строго ограничивает вас в их выборе (char, short, int и long). Можно, впрочем, определить и свой целочисленный класс, никто не запрещает. В main() реализуйте класс frac<char> и используйте его при разработке калькулятора с четырьмя функциями. Этому классу требуется меньше памяти, чем frac<int>, но с его помощью невозможно выполнять деление больших чисел.

8. Добавьте класс исключений к программе ARROVER из главы 8, чтобы индексы, выходящие за пределы массива, вызывали генерацию исключения. Блок-ловушка может выводить пользователю сообщение об ошибке.

9. Измените программу из предыдущего упражнения таким образом, чтобы в сообщение об ошибке входила информация о значении индекса, приведшего к сбою.

10. Есть разные мнения на тему того, когда целесообразно использовать механизм исключений. Обратимся к программе ENGLERR из главы 12 «Потоки и файлы». Надо ли в виде исключений оформлять ошибки, которые делает пользователь при вводе данных? Для этого упражнения будем считать, что надо. Добавьте класс исключений к классу Distance в указанной программе (см. также примеры XDIST и XDIST2 в этой главе). Сгенерируйте исключения во всех местах, где ENGLERR выводила сообщение об ошибке. Для выявления конкретной причины исключения (вместо дюймов введены какие-то символы, значение дюймов выходит за область допустимых

значений и т. д.). Кроме того, исключения должны генерироваться в качестве реакции на ошибки, возникающие в функции isint() (ничего не было введено, введено слишком много разрядов значений, введен символ вместо числа, число выходит за область допустимых значений). Вопрос к вам: если функция isint() сгенерировала исключение, может ли она оставаться независимой?

Можно зациклить блок повторных попыток и блок-ловушку (создать цикл do), чтобы после обработки исключения программа продолжала работу, предлагая пользователю повторно ввести данные.

Если еще немного подумать, то может возникнуть желание генерировать исключение в конструкторе с двумя аргументами (в том случае, если программист инициализирует Distance значениями, выходящими за область допустимых).

11. За основу возьмите программу STRPLUS из главы 8. Добавьте класс исключений, генерируйте исключения в конструкторе с одним аргументом в случае, если строка инициализации слишком длинная. Генерируйте еще одно исключение в перегруженном операторе «+», если результат конкатенации оказывается слишком длинным. Сообщайте пользователю о том, какая именно ошибка произошла.

12. Иногда проще всего бывает использовать механизм исключений, если создать новый класс, чьим компонентом является класс исключений. Попробуйте проделать такую операцию с классом, использующим исключения для обработки файловых ошибок. Создайте класс dofile, включающий в себя класс исключений и методы чтения, и записи файлов. Конструктор этого класса в качестве аргумента может брать имя файла, а действием конструктора, соответственно, может быть открытие этого файла. Можно сделать так, чтобы метод переставлял указатель позиции в начало файла. Используйте программу REWERR из главы 12 в качестве примера, напишите секцию main(), в которой сохранилась бы функциональность прежней версии, но использовались бы методы класса dofile.

Глава 15 Стандартная библиотека шаблонов (STL)

- Введение в STL
- Алгоритмы
- Последовательные контейнеры
- Итераторы
- Специальные итераторы
- Ассоциативные контейнеры
- Хранение пользовательских объектов
- Функциональные объекты

Большинство компьютеров предназначено для обработки информации. В качестве данных могут выступать самые разные виды характеристик реального мира: это может быть досье на работников, информация об имеющихся на складе запасах, текстовый документ, результат научных экспериментов и т. д. Что бы данные собой ни представляли, хранятся и обрабатываются они примерно одинаковыми способами. В университетские учебные планы по специальности «Информатика» обычно входит курс «Структуры данных и алгоритмы». Термин *структура данных* говорит о том, как хранится информация в памяти компьютера, а *алгоритм* — как эта информация обрабатывается.

Классы C++ предоставляют прекрасный механизм для создания библиотеки структур данных. В прошлом производители компиляторов и разные сторонние разработчики ПО предлагали на рынке библиотеки *классов-контейнеров* для хранения и обработки данных. Теперь же в стандарт C++ входит собственная встроенная библиотека классов-контейнеров. Она называется Стандартной библиотекой шаблонов (в дальнейшем мы будем употреблять сокращение STL) и разработана Александром Степановым и Менг Ли из фирмы Hewlett Packard. STL — это часть Стандартной библиотеки классов C++, которая может использоваться в качестве этакой повседневной палочки-выручалочки для хранения и обработки данных.

В данной главе описывается принцип построения STL и работа с ней. Тема большая и сложная, и мы не будем описывать все, что входит в библиотеку, поскольку одно это потребовало бы написания отдельной увесистой книги (по STL написано довольно много учебников и книг. См. приложение З «Библиография»). Мы ограничимся здесь тем, что представим STL и приведем примеры наиболее часто используемых алгоритмов и контейнеров.

Введение в STL

В STL содержится несколько основных сущностей. Три наиболее важные из них — это контейнеры, алгоритмы и итераторы.

Контейнер — это способ организации хранения данных. В предыдущих главах нам уже встречались некоторые контейнеры, такие, как стек, связный список, очередь. Еще один контейнер — это массив, но он настолько тривиален и популярен, что встроен в C++ и большинство других языков программирования. Контейнеры бывают самые разнообразные, и в STL включены наиболее полезные из них. Контейнеры STL подключаются к программе с помощью шаблонных классов, а значит, можно легко изменить тип хранимых в них данных.

Под *алгоритмами* в STL подразумевают процедуры, применяемые к контейнерам для обработки их данных различными способами. Например, есть алгоритмы сортировки, копирования, поиска и объединения. Алгоритмы представлены в STL в виде шаблонных функций. Однако они не являются методами классов-контейнеров. Наоборот, это совершенно независимые функции. На самом деле, одной из самых привлекательных черт STL является универсальность ее алгоритмов. Их можно использовать не только в объектах классов-контейнеров, но и в обычных массивах и даже в собственных контейнерах. (Контейнеры, тем не менее, содержат методы для выполнения некоторых специфических задач.)

Итераторы — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы — ключевая часть всего STL, поскольку они связывают алгоритмы с контейнерами. Их можно представить себе в виде кабеля, связывающего колонки вашей стереосистемы или компьютер с его периферией.

На рис. 15.1 показаны все три компонента STL. В этом параграфе мы как раз их и обсудим более детально. В следующих параграфах мы проиллюстрируем изложенную концепцию примерами программ.

Контейнеры

Как мы уже говорили, контейнеры представляют собой различные структуры для хранения данных. При этом не имеет значения, какие именно данные хранятся, будь то некоторые базовые типы, такие, как int, float и т. п., или же объекты классов. STL включает в себя семь основных типов контейнеров и еще три производных типа. Зачем же нам столько типов контейнеров? Почему нельзя

использовать обычный массив во всех случаях, когда нужно хранить данные? Ответ таков: неэффективно. Работать с массивом во многих ситуациях бывает неудобно — медленно и вообще затруднительно.

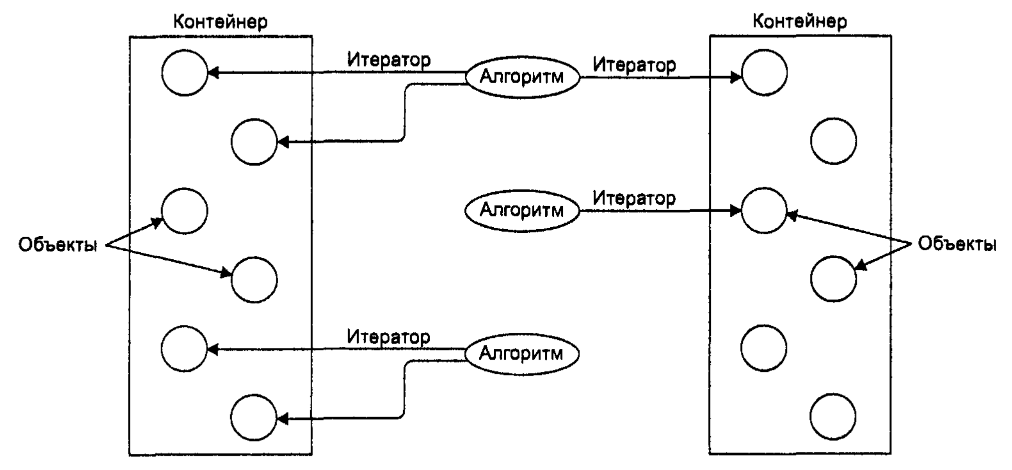


Рис. 15.1. Контейнеры, алгоритмы и итераторы

Контейнеры STL подразделяются на две категории: *последовательные* и *ассоциативные*. Среди последовательных выделяют *векторы*, *списки* и *очереди с двусторонним доступом*. Среди ассоциативных — *множества*, *мультимножества*, *отображения* и *мультиотображения*. Кроме того, наследниками последовательных выступают еще несколько специализированных контейнеров: *стек*, *очередь* и *приоритетная очередь*. Рассмотрим пункты этой классификации более подробно.

Последовательные контейнеры

В последовательных контейнерах данные хранятся подобно тому, как дома стоят на улице — в ряд. Каждый элемент связывается с другими посредством номера своей позиции в ряду. Все элементы, кроме конечных, имеют по одному соседу с каждой стороны. Примером последовательного контейнера является обычный массив.

Проблема с массивами заключается в том, что их размеры нужно указывать при компиляции, то есть в исходном коде. К сожалению, во время написания программы обычно не бывает известно, сколько элементов потребуется записать в массив. Поэтому, как правило, рассчитывают на худший вариант, и размер массива указывают «с запасом». В итоге при работе программы выясняется, что либо в массиве остается очень много свободного места, либо все-таки не хватает ячеек, а это может привести к чему угодно, вплоть до зависания программы. В STL для борьбы с такими трудностями существует контейнер *vector*.

Но с массивами есть еще одна проблема. Допустим, в массиве хранятся данные о работниках, и вы отсортировали их по алфавиту в соответствии с фамилиями. Теперь, если нужно будет добавить сотрудника, фамилия которого начинается с буквы Л, то придется сдвинуть всех работников с фамилиями на М — Я,

чтобы освободить место для вставки нового значения. Все это может занимать довольно много времени. В STL имеется контейнер *список*, который основан на идее связного списка и который решает данный вопрос. Вспомните программу LINKLIST из главы 10 «Указатели», там очень просто осуществлялась вставка данных в связный список с помощью перестановки нескольких указателей.

Третьим представителем последовательных контейнеров является *очередь с двусторонним доступом*. Это комбинация стека и обычной очереди. Стек, как вы помните, работает по принципу LIFO (последний вошел — первый вышел). То есть и ввод, и вывод данных в стеке производится «сверху». В очереди же, напротив, используется принцип FIFO (первый вошел — первый вышел): данные поступают «сверху», а выходят «снизу». Очередь с двусторонним доступом, надо полагать, использует комбинированный порядок обмена данных. И вносить значения в очередь с двусторонним доступом, и выводить из нее можно с обеих сторон. Это довольно гибкий инструмент, он ценен не только сам по себе, но может служить базой для стеков очередей, в чем мы вскоре сможем убедиться.

В табл. 15.1 сведены характеристики последовательных контейнеров STL. Для сравнения приводится и обычный массив.

Таблица 15.1. Основные последовательные контейнеры

Контейнер	Характеристика	Плюсы/минусы
Обычный массив	Фиксированный размер	Скоростной случайный доступ (по индексу)
		Медленная вставка или изъятие данных из середины
		Размер не может быть изменен во время работы программы
Вектор	Перераспределяемый, расширяемый массив	Скоростной случайный доступ (по индексу)
		Медленная вставка или изъятие данных из середины
		Скоростная вставка или изъятие данных из хвоста
Список	Аналогичен связному списку	Скоростная вставка или изъятие данных из любого места
		Быстрый доступ к обоим концам
		Медленный случайный доступ
Очередь с двусторонним доступом	Как вектор, но доступ с обоих концов	Скоростной случайный доступ
		Медленная вставка или изъятие данных из середины
		Скоростная вставка и изъятие данных из хвоста или головы

Реализация на практике контейнера STL не представляет особого труда. Во-первых, нужно включить в программу соответствующий заголовочный файл. Передача информации о том, какие типы объектов будут храниться, производится в виде передачи параметра в шаблон. Например:

```
vector<int> aVect;            //создать вектор целых чисел (типа int)
```

или

```
list<airtime> departure_list; //создать список типа airtime
```

Обратите внимание: в STL не нужно специфицировать размеры контейнеров. Они сами заботятся о размещении своих данных в памяти.

Ассоциативные контейнеры

Ассоциативный контейнер — это уже несколько иная организация данных. Данные располагаются не последовательно, доступ к ним осуществляется посредством *ключей*. Ключи — это просто номера строк, они обычно используются для выстраивания хранящихся элементов в определенном порядке и модифицируются контейнерами автоматически. Примером может служить обычный орфографический словарь, где слова сортируются по алфавиту. Вы просто вводите нужное слово (значение ключа), например «арбуз», а контейнер конвертирует его в адрес этого элемента в памяти. Если известен ключ, доступ к данным осуществляется очень просто.

В STL имеется два типа ассоциативных контейнеров: *множества* и *отображения*. И те и другие контейнеры хранят данные в виде *дерева*, что позволяет осуществлять быструю вставку, удаление и поиск данных. Множества и отображения являются очень удобными и при этом достаточно универсальными инструментами, которые могут применяться в очень многих ситуациях. Но осуществлять сортировку и выполнять другие операции, требующие случайного доступа к данным, неудобно.

Множества проще и, возможно, из-за этого более популярны, чем отображения. Множество хранит набор элементов, содержащих *ключи* — атрибуты, используемые для упорядочивания данных. Например, множество может хранить объекты класса person, которые упорядочиваются в алфавитном порядке. В качестве ключа при этом используется значение name. При такой организации данных можно очень быстро найти нужный объект, осуществляя поиск по имени объекта (ищем объект класса person по атрибуту name). Если в множестве хранятся значения одного из базовых типов, таких, как int, ключом является сам элемент. Некоторые авторы называют ключом весь объект, хранящийся в множестве, но мы будем употреблять термин *ключевой объект*, чтобы подчеркнуть, что атрибут, используемый для упорядочивания (ключ), — это не обязательно весь элемент данных.

Отображение хранит пары объектов. Один кортеж в такой «базе данных» состоит из двух «атрибутов»: ключевой объект и целевой (ассоциированный) объект. Этот инструмент часто используется в качестве контейнера, несколько напоминающего массив. Разница лишь в том, что доступ к данным осуществляется не по номеру элемента, а по специальному индексу, который сам по себе может быть произвольного типа. То есть ключевой объект служит индексом, а целевой — значением этого индекса.

Контейнеры *отображение* и *множество* разрешают сопоставлять только один ключ данному значению. Это имеет смысл в таких, например, случаях, как хранение списка работников, где каждому имени сопоставляется уникальный идентификационный номер. С другой стороны, *мультиотображения* и *мультимножества* позволяют хранить несколько ключей для одного значения. Например, слову «ключ» в толковом словаре может быть сопоставлено несколько статей.

В табл. 15.2 собраны все ассоциативные контейнеры, имеющиеся в STL.

Таблица 15.2. Основные ассоциативные контейнеры

Контейнер	Характеристики
Множество	Хранит только ключевые объекты. Каждому значению сопоставлен один ключ
Мультимножество	Хранит только ключевые объекты. Одному значению может быть сопоставлено несколько ключей
Отображение	Ассоциирует ключевой объект с объектом, хранящим значение (целевым). Одному значению сопоставлен один ключ
Мультиотображение	Ассоциирует ключевой объект с объектом, хранящим значение (целевым). Одному значению может быть сопоставлено несколько ключей

Создаются ассоциативные контейнеры примерно так же, как и последовательные:

```
set<int> intSet;            //создает множество значений int
```

или

```
multiset<employee> machinists; //создает мультимножество
                               //значений класса employee
```

Методы

Алгоритмы — это рабочие лошадки STL, выполняющие сложные операции типа сортировки и поиска. Однако для выполнения более простых операций, специфичных для конкретного контейнера, требуются методы.

В табл. 15.3 представлены некоторые наиболее часто используемые методы, имена и назначение которых общие для большинства классов контейнеров.

Таблица 15.3. Некоторые методы, общие для всех контейнеров

Имя	Назначение
size()	Возвращает число элементов в контейнере
empty()	Возвращает true, если контейнер пуст
max_size()	Возвращает максимально допустимый размер контейнера
begin()	Возвращает итератор на начало контейнера (итерации будут производиться в прямом направлении)
end()	Возвращает итератор на последнюю позицию в контейнере (итерации в прямом направлении будут закончены)
rbegin()	Возвращает реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)
rend()	Возвращает реверсивный итератор на начало контейнера (итерации в обратном направлении будут завершены)

Существует еще множество методов, которые применяются в конкретных контейнерах или категориях контейнеров. Вы узнаете больше о них, заглянув в приложение Е «Алгоритмы и методы STL». Там приведена таблица, содержащая методы и контейнеры, к которым данные методы относятся.

Адаптеры контейнеров

Специализированные контейнеры можно создавать из базовых (приведенных выше) с помощью конструкции, называющейся *адаптером контейнера.* Они обладают более простым интерфейсом, чем обычные контейнеры. Специализированные контейнеры, реализованные в STL, это *стеки*, *очереди* и *приоритетные очереди.* Как уже отмечалось, для стека характерен доступ к данным только с одного конца, его можно сравнить со стопкой книг. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В приоритетной очереди данные проталкиваются спереди в произвольном порядке, а выталкиваются в строгом соответствии с величиной хранящегося значения. Приоритет имеют данные с наибольшим значением. Таким образом, приоритетная очередь автоматически сортирует хранящуюся в ней информацию.

Стеки, очереди и приоритетные очереди могут создаваться из разных последовательных контейнеров, хотя, например, очередь с двусторонним доступом используется тоже довольно часто. В табл. 15.4 показаны абстрактные типы и последовательные контейнеры, использующиеся для их реализации.

Таблица 15.4. Контейнеры, реализуемые с помощью адаптеров

Контейнер	Реализация	Характеристики
Стек	Реализуется как вектор, список или очередь с двусторонним доступом	Проталкивание и выталкивание данных только с одного конца
Очередь	Реализуется как список или очередь с двусторонним доступом	Проталкивание с одного конца, выталкивание — с другого
Приоритетная очередь	Реализуется как вектор или очередь с двусторонним доступом	Проталкивание с одного конца в случайном порядке, выталкивание — упорядоченное, с другого конца

Для практического применения этих классов необходимо использовать как бы шаблон в шаблоне. Например, пусть имеется объект типа стек, содержащий значения int, порожденный классом «очередь с двусторонним доступом» (deque):

```
stack< deque<int> > aStak;
```

Деталь, на которую стоит обратить внимание при описании этого формата, это пробелы, которые необходимо ставить между двумя закрывающими угловыми скобками. Помните об этом, потому что такое выражение

```
stack<deque<int>> aStak;
```

приведет к синтаксической ошибке — компилятор интерпретирует >> как оператор.

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Как уже говорилось выше, алгоритмы в STL не являются методами классов и даже не являются дружественными функциями по отношению к контейнерам, как было когда-то. В нынешнем стандарте языка алгоритмы — это независимые шаблонные функции. Их можно использовать при

работе как с обычными массивами C++, так и с вашими собственными классами-контейнерами (предполагается, что в класс включены уже некоторые базовые функции).

В табл. 15.5 показаны некоторые популярные алгоритмы. В дальнейшем мы рассмотрим и другие примеры. А в приложении Е можно найти таблицу, в которой содержатся практически все алгоритмы STL.

Таблица 15.5. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
find	Возвращает первый элемент с указанным значением
count	Считает количество элементов, имеющих указанное значение
equal	Сравнивает содержимое двух контейнеров и возвращает true, если все соответствующие элементы эквивалентны
search	Ищет последовательность значений в одном контейнере, которая соответствует такой же последовательности в другом
copy	Копирует последовательность значений из одного контейнера в другой (или в другое место того же контейнера)
swap	Обменивает значения, хранящиеся в разных местах
iter_swap	Обменивает последовательности значений, хранящиеся в разных местах
fill	Копирует значение в последовательность ячеек
sort	Сортирует значения в указанном порядке
merge	Комбинирует два сортированных диапазона значений для получения наибольшего диапазона
accumulate	Возвращает сумму элементов в заданном диапазоне
for_each	Выполняет указанную функцию для каждого элемента контейнера

Допустим, вы создаете массив типа int со следующими данными:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

Применим алгоритм sort() для сортировки массива:

```
sort(arr, arr+8);
```

где arr — это адрес начала массива, arr+8 — адрес конца (элемент, располагающийся позади последнего в массиве).

Итераторы

Итераторы — это сущности, напоминающие указатели. Они используются для получения доступа к отдельным данным (которые обычно называются *элементами*) в контейнере. Они часто используются для последовательного продвижения по контейнеру от элемента к элементу (этот процесс называется *итерацией*). Итераторы можно инкрементировать с помощью обычного оператора ++, после выполнения которого итератор передвинется на следующий элемент. Косвенность со ссылок снимается оператором *, после чего можно получить значение элемента, на который ссылается итератор. В STL итератор представляет собой объект класса iterator.

Для разных типов контейнеров используются свои итераторы. Всего существует три основных класса итераторов: прямые, двунаправленные и со случайным доступом. *Прямой итератор* может проходить по контейнеру только в прямом направлении, что и указано в его названии. Проход осуществляется поэлементный. Работать с ним можно, используя ++. Такой итератор не может двигаться в обратном направлении и не может быть поставлен в произвольное место контейнера. *Двунаправленный итератор*, соответственно, может передвигаться в обоих направлениях и реагирует как на ++, так и на --. *Итератор со случайным доступом* может и двигаться в обоих направлениях, и перескакивать на произвольное место контейнера. Можно приказать ему получить доступ к позиции 27, например.

Есть два специализированных вида итераторов. Это *входной итератор*, который может «указывать» на устройство ввода (cin или даже просто входной файл) и считывать последовательно элементы данных в контейнер, и *выходной итератор*, который, соответственно, указывает на устройство вывода (cout) или выходной файл и выводит элементы из контейнера.

В то время как значения прямых, двунаправленных итераторов и итераторов со случайным доступом могут быть сохранены, значения входных и выходных операторов сохраняться не могут. Это имеет смысл, ибо первые три оператора все-таки указывают на некоторый адрес в памяти, тогда как входные и выходные итераторы указывают на устройства ввода/вывода, для которых хранить какой-то «указатель» невозможно. В табл. 15.6 показаны характеристики различных типов итераторов.

Таблица 15.6. Характеристики итераторов

Тип итератора	Запись/Чтение	Хранение значения	Направление	Доступ
Со случайным доступом	Запись и чтение	Возможно	Оба направления	Случайный
Двунаправленный	Запись и чтение	Возможно	Оба направления	Линейный
Прямой	Запись и чтение	Возможно	Только прямое	Линейный
Выходной	Только запись	Невозможно	Только прямое	Линейный
Входной	Только чтение	Невозможно	Только прямое	Линейный

Возможные проблемы с STL

Шаблонные классы STL достаточно сложны, при их обработке компилятору приходится несладко. Не все компиляторы выдерживают. Рассмотрим некоторые возможные проблемы.

Во-первых, иногда бывает сложно отыскать ошибку, поскольку компилятор сообщает, что она произошла где-то в глубинах заголовочного файла, в то время как на самом деле она произошла на самом видном месте в исходном файле. Вам придется перелопатить уйму кода, комментируя одну строчку за другой, а диагностировать ошибку, возможно, так и не удастся.

Прекомпиляция заголовочных файлов, которая невероятно ускоряет процесс компиляции программы, может вызвать определенные проблемы при использовании STL. Если вам кажется, что что-то не так, попробуйте отключить прекомпиляцию.

STL может генерировать разные забавные сообщения. Например, «При преобразовании могут потеряться значащие разряды!» — это просто хит. В принципе, эти ошибки довольно безобидны, не стоит обращать на них внимания. Если они вас сильно раздражают, можно их отключить вообще.

Несмотря на мелкие претензии, которые можно предъявить к STL, это удивительно мощный и гибкий инструмент. Все ошибки можно исключить на этапе компиляции, а не во время работы программы. Алгоритмы и контейнеры имеют очень содержательный и устойчивый интерфейс. То, что работает в применении к одному контейнеру или алгоритму, будет, скорее всего, работать и в применении к другим (конечно, при условии их правильного использования).

Этот небольшой обзор, наверное, в большей мере поставил вопросы, чем дал ответы. Далее мы постараемся разобрать STL более детально, чтобы стало понятно, что с этим добром делать.

Алгоритмы

Алгоритмы STL выполняют различные операции над наборами данных. Они были разработаны специально для контейнеров, но их замечательным свойством является то, что они применимы и к обычным массивам C++. Это может сильно упростить работу с массивами. К тому же изучение алгоритмов для контейнеров поможет усвоить общие принципы подобной обработки данных, безотносительно к контейнерам и STL (сами алгоритмы можно найти в приложении Е).

Алгоритм find ()

Этот алгоритм ищет первый элемент в контейнере, значение которого равно указанному. В примере FIND показано, как нужно действовать, если мы хотим найти значение в массиве целых чисел.

Листинг 15.1. Программа FIND[1]

```
// find.cpp
// найти первый объект, значение которого равно данному
#include <iostream>
#include <algorithm>          //для find()
using namespace std;

int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };

int main()
  {
  int* ptr;
  ptr = find(arr, arr+8, 33);  //найти первое вхождение 33
  cout << "Первый объект со значением 33 найден в позиции "
       << (ptr-arr) << endl;
  return 0;
  }
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

Результаты программы:

```
Первый объект со значением 33 найден в позиции 2
```

Как и всегда, первый элемент в массиве имеет индекс 0, а не 1, поэтому число 33 было найдено в позиции 2, а не 3.

Заголовочные файлы

В эту программу включен заголовочный файл ALGORITHM. (Заметим, что, как и в других заголовочных файлах Стандартной библиотеки C++, расширения типа .H не пишут.) В этом файле содержатся объявления алгоритмов STL. Другие заголовочные файлы используются для контейнеров и для иных целей. Если вы используете старые версии STL, вам может понадобиться заголовочный файл с немного другим именем: ALGO.H.

Диапазоны

Первые два аргумента алгоритма find() определяют диапазон просматриваемых элементов. Значения задаются итераторами. В данном примере мы использовали значения обычных указателей C++, которые, в общем-то, являются частным случаем итераторов.

Первый параметр — это итератор (то есть в данном случае — указатель) первого значения, которое нужно проверять. Второй параметр — итератор последней позиции (на самом деле он указывает на следующую позицию за последним нужным значением). Так как всего в нашем массиве 8 элементов, это значение равно первой позиции, увеличенной на 8. Это называется «значение после последнего».

Используемый при этом синтаксис является вариацией на тему обычного синтаксиса цикла for в C++:

```
for(int j=0; j<8; j++)      //от 0 до 7
   {
   if(arr[j] == 33)
      {
      cout << " Первый объект со значением 33 найден в позиции " << j << endl;
      break;
      }
   }
```

В программе FIND алгоритм find() не спасает вас от необходимости писать цикл for. В более интересных случаях, чем этот пример, алгоритмы могут спасти и не от такого. Очень сложные и длинные коды, бывает, заменяются алгоритмами.

Алгоритм count ()

Взглянем на другой алгоритм — count(). Он подсчитывает, сколько элементов в контейнере имеют данное значение. В примере COUNT это продемонстрировано.

Листинг 15.2. Программа COUNT

```
// count.cpp
// считает количество объектов, имеющих данное значение
#include <iostream>
```

продолжение

Листинг 15.2 (*продолжение*)

```
#include <algorithm>             //для count()
using namespace std:

int arr[] = { 33. 22. 33. 44. 33. 55. 66. 77 }:

int main()
   {
   int n = count(arr. arr+8. 33):  //считать. сколько раз
                                   //встречается 33
   cout << "Число 33 встречается " << n << " раз(а) в массиве." << endl:
   return 0:
   }
```

На экране в результате мы увидим следующее:

```
Число 33 встречается 3 раз(а) в массиве.
```

Алгоритм sort()

Кажется, можно догадаться по названию, что делает этот алгоритм. Приведем пример применения его к массиву.

Листинг 15.3. Программа SORT

```
// sort.cpp
// сортирует массив целых чисел
#include <iostream>
#include <algorithm>
using namespace std:
                              // массив чисел
int arr[] = {45. 2. 22. -17. 0. -30. 25. 55}:

int main()
   {
   sort(arr. arr+8):          // сортировка

for(int j=0: j<8: j++)      // вывести отсортированный
                            //массив
      cout << arr[j] << ' ':
   cout << endl:
   return 0:
   }
```

Вот что мы видим на экране:

```
-30. -17. 0. 2. 22. 25. 45. 55
```

К некоторым вариациям на тему этого алгоритма мы еще вернемся несколько позднее.

Алгоритм search()

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Например, если алгоритм find() ищет указанное значение в одном контейнере, алгоритм search() ищет целую последовательность значений, заданную одним контейнером, в другом контейнере. Рассмотрим на примере.

Листинг 15.4. Программа SEARCH

```
// search.cpp
//Ищем последовательность, заданную одним контейнером,
// другом контейнере
#include <iostream>
#include <algorithm>
using namespace std;

int source[] =  { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };

int main()
   {
   int* ptr;
   ptr = search(source, source+9, pattern, pattern+3);
   if(ptr == source+9)              // если после последнего
      cout << "Совпадения не найдено\n";
   else
      cout << "Совпадение в позиции " << (ptr - source) << endl;
   return 0;
   }
```

Алгоритм просматривает последовательность 11, 12, 13, заданную массивом pattern, в массиве source. Видно, что эти числа подряд встречаются, начиная с четвертого элемента (позиция 3, считая с 0). Программа выводит на экран такой результат:

```
Совпадение в позиции 3
```

Если значение итератора ptr оказывается за пределами массива source, выводится сообщение о том, что совпадения не найдено.

Параметрами алгоритма search() и подобных не должны обязательно быть контейнеры одного типа. Исходный контейнер может быть, например, вектором STL, а маска поиска — обычным массивом. Такая универсальность — это одна из сильных сторон STL.

Алгоритм merge()

Этот алгоритм работает с тремя контейнерами, объединяя элементы двух из них в третий, целевой контейнер. Следующий пример показывает, как это делается.

Листинг 15.5. Программа MERGE

```
// merge.cpp
// соединение двух контейнеров в третий
#include <iostream>
#include <algorithm>          //для merge()
using namespace std;

int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];

int main()
```

продолжение ⇨

Листинг 15.5 (*продолжение*)

```
{                               //соединить src1 и src2 в dest
merge(src1, src1+5, src2, src2+3, dest);
for(int j=0; j<8; j++)      // вывести dest
   cout << dest[j] << ' ';
cout << endl;
return 0;
}
```

В итоге получается контейнер, содержимое которого таково:

```
1 2 3 3 4 5 6 8
```

Как видите, алгоритм объединения сохраняет порядок следования элементов, вплетая содержимое двух контейнеров в третий.

Функциональные объекты

Некоторым алгоритмам в качестве параметра требуются некие *функциональные объекты*. Для пользователя функциональный объект — это что-то вроде шаблонной функции. На самом же деле, это просто объект шаблонного класса, в котором имеется единственный метод: перегружаемая операция (). Звучит загадочно, но на практике применяется очень просто и безболезненно.

Допустим, вы хотите отсортировать массив чисел по возрастанию или по убыванию. Посмотрим, как это делается.

Листинг 15.6. Программа SORTEMP

```
// sortemp.cpp
// сортировка массива типа double по убыванию,
// используется функциональный объект greater<>()
#include <iostream>
#include <algorithm>                 //для sort()
#include <functional>                //для greater<>
using namespace std;
                                     // массив double
double fdata[] = { 19.2, 87.4, 33.6, 55.0, 11.5, 42.2 };

int main()
   {                        // сортировка значений double
   sort( fdata, fdata+6, greater<double>() );

   for(int j=0; j<6; j++)  // вывести отсортированный массив
      cout << fdata[j] << ' ';
   cout << endl;
   return 0;
   }
```

Алгоритм sort() вообще-то обычно сортирует по возрастанию, но использование функционального объекта greater<>() в качестве третьего аргумента sort() изменяет режим сортировки:

```
87.4 55 42.2 33.6 19.2 11.5
```

Кроме сравнения, есть функциональные объекты для арифметических и логических операций. Мы еще обратимся к этой теме в последнем разделе главы.

Пользовательские функции вместо функциональных объектов

Дело в том, что функциональные объекты могут работать только с базовыми типами C++ и с классами, для которых определены соответствующие операторы (+, -, <, == и т. д.). Это не всегда удобно, поскольку иногда приходится работать с типами данных, для которых такое условие не выполняется. В этом случае можно определить собственную пользовательскую функцию для функционального объекта. Например, оператор < не определен для обычных строк char*, но можно написать функцию, выполняющую сравнение, и использовать ее адрес (имя) вместо функционального объекта. Программа SORTCOM показывает, как отсортировать массив строк char*.

Листинг 15.7. Программа SORTCOM

```
// sortcom.cpp
// сортировка массива строк с помощью пользовательской //функции сортировки
#include <iostream>
#include <string>                    // для strcmp()
#include <algorithm>
using namespace std;
                                     // массив строк
char* names[] = { "Сергей", "Татьяна", "Елена",
                  "Дмитрий", "Михаил", "Владимир" };

bool alpha_comp(char*, char*);       // объявление

int main()
   {
   sort(names, names+6, alpha_comp); // сортировка строк

   for(int j=0; j<6; j++)      // вывод отсортированных строк
      cout << names[j] << endl;
   return 0;
   }

bool alpha_comp(char* s1, char* s2)  // возвращает true
                                     // если s1<s2
   {
   return ( strcmp(s1, s2)<0 ) ? true : false;
   }
```

Третьим параметром алгоритма sort() в данном случае является адрес функции alpha_comp(), сравнивающей две строки типа char* и возвращающей true или false в зависимости от того, правильно ли для них осуществлена лексикографическая сортировка. Если неправильно, строки меняются местами. В программе используется библиотечная функция C под названием strcmp(), которая возвращает отрицательное значение в случае, если первый аргумент меньше второго. Результат программы вполне ожидаем:

```
Владимир
Дмитрий
Елена
Михаил
```

```
Сергей
Татьяна
```

Вообще-то писать собственные функциональные объекты для работы с текстом вовсе не обязательно. Используйте класс string из стандартной библиотеки и будьте счастливы. Ибо в нем есть встроенные функциональные объекты, такие, как less<>() и greater<>().

Добавление _if к аргументам

Некоторые аргументы имеют версии с окончанием _if. Им требуется дополнительный параметр, который называется *предикатом* и является функциональным объектом или функцией. Например, алгоритм find() находит все элементы, равные указанному значению. Можно написать функцию, которая работает с алгоритмом find_if()и находит элементы с какими-либо дополнительными параметрами.

В нашем примере используются объекты класса string. Алгоритму find_if() передается пользовательская функция isDon(), чтобы можно было искать первую строку в массиве, имеющую значение «Дмитрий». Приведем листинг примера.

Листинг 15.8. Программа FIND_IF

```
// find_if.cpp
// ищет в массиве типа string первое вхождение слова
//«Дмитрий»
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
//-----------------------------------------------------------
bool isDon(string name)      // возвращает true, если
                             //name=="Дмитрий"
   {
   return name == "Дмитрий";
   }
//-----------------------------------------------------------
string names[] = { "Сергей", "Татьяна", "Елена", "Дмитрий", "Михаил", "Владимир" };
int main()
   {
   string* ptr;
   ptr = find_if( names, names+5, isDon );

   if(ptr==names+5)
      cout << "Дмитрия нет в списке.\n";
   else
      cout << "Дмитрий записан в позиции "
        << (ptr-names)
        << " в списке.\n";
   return 0;
   }
```

Так как слово «Дмитрий» действительно встречается в массиве, то результат работы программы будет такой:

```
Дмитрий записан в позиции 2 в списке.
```

Адрес функции isDon() — третий аргумент алгоритма find_if(), а первые два, как обычно, задают диапазон поиска от начала до «после последнего» элемента массива.

Алгоритм find_if() применяет функцию isDon() к каждому элементу из диапазона. Если isDon() возвращает true для какого-либо элемента, то find_if() возвращает значение итератора этого элемента. В противном случае возвращается указатель на адрес «после последнего» элемента массива.

_if-версии имеются и у других алгоритмов, например count(), replace(), remove().

Алгоритм for_each ()

Этот алгоритм позволяет выполнять некое действие над каждым элементом в контейнере. Вы пишете собственную функцию, чтобы определить, какое именно действие следует выполнять. Эта ваша функция не имеет права модифицировать данные, но она может их выводить или использовать их значения в своей работе.

Вот пример, в котором for_each() используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран. Мы пишем функцию in_to_cm(), которая просто умножает значение на 2,54, и передаем адрес этой функции в качестве третьего параметра алгоритма.

Листинг 15.9. Программа FOR_EACH

```
// for_each.cpp
// for_each() используется для вывода элементов массива. //переведенных из дюймов в сантиметры
#include <iostream>
#include <algorithm>
using namespace std;

void in_to_cm(double);   // объявление

int main()
   {         // массив значений в дюймах
   double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
             // вывод в виде сантиметров
   for_each(inches, inches+5, in_to_cm);
   cout << endl;
   return 0;
   }

void in_to_cm(double in) // перевод и вывод в сантиметрах
   {
   cout << (in * 2.54) << ' ';
   }
```

Результаты программы выглядят так:

```
8.89 15.748 2.54 32.385 10.9982
```

Алгоритм transform()

Этот алгоритм тоже делает что-то с каждым элементом контейнера, но еще и помещает результат в другой контейнер (или в тот же). Опять же, пользователь-

ская функция определяет, что именно делать с данными, причем тип возвращаемого ею результата должен соответствовать типу целевого контейнера. Наш пример, в общем-то, идентичен примеру FOR_EACH за единственным исключением: вместо вывода на экран, функция in_to_cm() выводит значения сантиметров в новый массив centi[], затем главная программа выводит содержимое этого массива. Приведем листинг программы.

Листинг 15.10. Программа TRANSFO

```
// transfo.cpp
// transform() используется для перевода значений из дюймов //в сантиметры
#include <iostream>
#include <algorithm>
using namespace std;

int main()
   {                         // массив дюймов
   double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
   double centi[5];
   double in_to_cm(double);  // прототип
                             // перевод в массив centi[]
   transform(inches, inches+5, centi, in_to_cm);

   for(int j=0; j<5; j++)    // вывод массива centi[]
      cout << centi[j] << ' ';
   cout << endl;
   return 0;
   }

double in_to_cm(double in)  // перевод дюймов в сантиметры
   {
   return (in * 2.54);       // вернуть результат
   }
```

Результаты программы такие же, как в FOR_EACH.

Итак, мы рассмотрели некоторые алгоритмы STL. Есть еще множество других, но, кажется, из приведенных примеров вполне понятно, какие они бывают и как с ними следует обращаться.

Последовательные контейнеры

Как уже отмечалось, есть две основные категории контейнеров: последовательные и ассоциативные. В этом разделе мы рассмотрим три последовательных контейнера (векторы, списки и очереди с двусторонним доступом), детально обсуждая, как работает каждый из них и какие в них имеются методы. Мы еще почти ничего не знаем об итераторах, поэтому будем избегать некоторых операций, которые без этих знаний не будут понятны. Про итераторы читайте в следующем разделе.

Каждый пример в следующих разделах будет представлять несколько методов описываемого контейнера. Тем не менее следует помнить, что разные виды контейнеров используют методы с одинаковыми именами и характеристиками, поэтому, например, метод push_back() для векторов будет ничуть не хуже работать со списками и очередями.

Векторы

Векторы — это, так сказать, умные массивы. Они занимаются автоматическим размещением себя в памяти, расширением и сужением своего размера по мере вставки или удаления данных. Векторы можно использовать в какой-то мере как массивы, обращаясь к элементам с помощью привычного оператора []. Случайный доступ выполняется очень быстро в векторах.

Также довольно быстро осуществляется добавление (или *проталкивание*) новых данных в конец вектора. Когда это происходит, размер вектора автоматически увеличивается для того, чтобы было куда положить новое значение.

Методы push_back(), size() и operator[]

Наш первый пример касается самых общих векторных операций.

Листинг 15.11. Программа VECTOR

```
// vector.cpp
// Демонстрация push_back(), operator[], size()
#include <iostream>
#include <vector>
using namespace std;

int main()
   {
   vector<int> v;           // создать вектор типа int

   v.push_back(10);         // внести данные в конец вектора
   v.push_back(11);
   v.push_back(12);
   v.push_back(13);

   v[0] = 20;               // заменить новыми значениями
   v[3] = 23;

   for(int j=0; j<v.size(); j++)  // вывести содержимое
      cout << v[j] << ' ';        // 20 11 12 23
   cout << endl;
   return 0;
   }
```

Для создания вектора v используется штатный конструктор вектора без параметров. Как с любыми контейнерами STL, для задания типа переменных, которые будут храниться в векторе, используется шаблонный формат (в данном случае это тип int). Мы не определяем размер контейнера, поэтому вначале он равен 0.

Метод push_back() вставляет значение своего аргумента в конец вектора (конец располагается там, где находится самый большой индекс). Начало вектора (элемент с индексом 0), в отличие от списков и очередей, не может использоваться для вставки новых элементов. Здесь мы проталкиваем значения 10, 11, 12 и 13 таким образом, что v[0] содержит 10, v[1] содержит 11, v[2] содержит 12, и v[3] содержит 13.

Как только в векторе появляются какие-либо данные, к ним сразу может быть получен доступ с помощью перегруженного оператора []. Точно тот же

прием, что и при работе с массивами, как видите. Очень удобно. Этот оператор мы использовали в нашем примере, чтобы заменить первый элемент с 10 на 20, а последний — с 13 на 23. Таким образом, результат программы получился следующим:

```
20 11 12 23
```

Метод size() возвращает текущее число элементов, содержащихся в контейнере. Для программы VECTOR это 4. Это значение используется в цикле for для вывода значений вектора на экран.

Есть еще один метод, max_size(), который мы не демонстрировали здесь. Он возвращает максимальный размер, до которого может расшириться контейнер. Это число зависит от типа хранимых в нем данных (понятно, что чем больше занимает один элемент данного типа, тем меньше элементов мы сможем хранить), типа контейнера и операционной системы. Например, на нашей системе max_size() возвращает 1 073 741 823 для целочисленного вектора.

Методы swap(), empty(), back() и pop_back()

Следующий пример, VECTCON, демонстрирует еще несколько методов и векторов.

Листинг 15.12. Программа VECTCON

```
// vectcon.cpp
// демонстрация конструкторов, swap(), empty(), back(), pop_back()
#include <iostream>
#include <vector>
using namespace std;

int main()
   {                                   // массив типа double
   double arr[] = { 1.1, 2.2, 3.3, 4.4 };

   vector<double> v1(arr, arr+4);     // инициализация вектора
                                      //массивом
   vector<double> v2(4);              // пустой вектор. Размер = 4

   v1.swap(v2);                       // обменять содержимое v1 и v2

   while( !v2.empty() )         // пока вектор не будет пуст,
      {
      cout << v2.back() << ' '; // вывести последний элемент
      v2.pop_back();            // и удалить его
      }                         // вывод: 4.4 3.3 2.2 1.1
   cout << endl;
   return 0;
   }
```

В этой программе мы использовали два новых конструктора векторов. Первый инициализирует вектор v1 значениями обычного массива C++, переданного ему в качестве аргумента. Аргументами этого конструктора являются указатели на начало массива и на элемент «после последнего». Во втором конструкторе вектор v2 инициализируется установкой его размера. Мы положили его равным 4. Но значение самого вектора при инициализации не передается. Оба вектора содержат данные типа double.

Метод swap() обменивает данные одного вектора на данные другого, при этом порядок следования элементов не изменяется. В этой программе в векторе v2 содержится только мусор какой-то, а данных не содержится, поэтому он обменивается на вектор v1. Результат работы программы таков:

```
4.4. 3.3. 2.2. 1.1
```

Метод back() возвращает значение последнего элемента вектора. Мы выводим его с помощью cout. Метод pop_back() удаляет последний элемент вектора.

Таким образом, при каждом прохождении цикла последний элемент будет иметь разные значения. (Немного удивительно, что pop_back() только удаляет последний элемент, но не возвращает его значение, как pop() при работе со стеком. Поэтому, в принципе, всегда нужно использовать pop_back() и back() в паре.)

Некоторые методы, например swap(), существуют и в виде алгоритмов. В таких случаях лучше предпочесть метод алгоритму. Работа с методом для конкретного контейнера обычно оказывается более эффективной. Иногда имеет смысл использовать и то, и другое. Например, такой подход можно использовать для обмена элементов двух контейнеров разных типов.

Методы insert() и erase()

Эти методы, соответственно, вставляют и удаляют данные при обращении к произвольной позиции контейнера. Их не рекомендуется использовать с векторами, поскольку в этом случае при каждой вставке или удалении приходится перекраивать всю структуру — разжимать, ужимать вектор... Все это не слишком эффективно. Тем не менее, когда скорость не играет очень большой роли, использовать эти методы и можно, и нужно. Следующий пример показывает, как это делается.

Листинг 15.13. Программа VECTINS

```
// vectins.cpp
// демонстрация методов insert(). erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
   {
   int arr[] = { 100. 110. 120. 130 }:  //массив типа int

   vector<int> v(arr. arr+4);           // инициализировать вектор
                                        //массивом

   cout << "\nПеред вставкой: ":
   for(int j=0: j<v.size(): j++)        // вывести все элементы
      cout << v[j] << ' ':

   v.insert( v.begin()+2. 115):         // вставить 115 в позицию 2

   cout << "\nПосле вставки:  ":
   for(j=0: j<v.size(): j++)            // вывести все элементы
```

продолжение

Листинг 15.13 (*продолжение*)

```
      cout << v[j] << ' ';

   v.erase( v.begin()+2 );            // удалить элемент со 2 позиции

   cout << "\nПосле удаления:
   for(j=0; j<v.size(); j++)          // вывести все элементы
      cout << v[j] << ' ';
   cout << endl;
   return 0;
   }
```

Метод insert() (по крайней мере, данная версия) имеет два параметра: будущее расположение нового элемента в контейнере и значение элемента. Прибавляем две позиции к результату выполнения метода begin(), чтобы перейти к элементу № 2 (третий элемент в контейнере, считая с нуля). Элементы от точки вставки до конца контейнера сдвигаются, чтобы было место для размещения вставляемого. Размер контейнера автоматически увеличивается на единицу.

Метод erase() удаляет элемент из указанной позиции. Оставшиеся элементы сдвигаются, размер контейнера уменьшается на единицу. Вот как выглядят результаты работы программы:

```
Перед вставкой:100110120130
После вставки:100110115120130
После удаления:100110120130
```

Списки

Контейнер STL под названием список представляет собой дважды связный список, в котором каждый элемент хранит указатель на соседа слева и справа. Контейнер содержит адрес первого и последнего элементов, поэтому доступ к обоим концам списка осуществляется очень быстро.

Методы push_front(), front() и pop_front

Наш первый пример работы со списками демонстрирует вставку, чтение и извлечение данных.

Листинг 15.14. Программа LIST

```
// list.cpp
// демонстрация методов push_front(), front(), pop_front()
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   list<int> ilist;

   ilist.push_back(30);      // вставка элементов в конец
   ilist.push_back(40);
   ilist.push_front(20);     // вставка элементов в начало
```

```
ilist.push_front(10);

int size = ilist.size(); // число элементов

for(int j=0; j<size; j++)
   {
   cout << ilist.front() << ' ';  // читать данные
                                  //из начала
   ilist.pop_front();    // извлечение данных из начала
   }
cout << endl;
return 0;
}
```

Поясним работу программы. Мы вставляем данные в конец и начало списка таким образом, чтобы при выводе на экран и удалении из начала сохранялся бы следующий порядок их следования:

```
10203040
```

Методы push_front(), front() и pop_front() аналогичны методам pop_back(), push_back() и back(), которые мы уже видели при работе с векторами.

Помните, что произвольный доступ к элементам списка использовать нежелательно, так как он осуществляется слишком неторопливо для нормальной обработки данных. Поэтому оператор [] даже не определен для списков. Если бы произвольный доступ был реализован, этому оператору пришлось бы проходить весь список, перемещаясь от ссылки к ссылке, вплоть до достижения нужного элемента. Это была бы очень медленная операция. Если вы считаете, что программе может потребоваться произвольный доступ к контейнеру, используйте векторы или очереди с двусторонним доступом.

Списки целесообразно использовать при частых операциях вставки и удаления где-либо в середине списка. При этих действиях использовать векторы и очереди нелогично, потому что все элементы над точкой вставки или удаления должны при этом быть сдвинуты. Что касается списков, то при тех же операциях изменяются лишь значения нескольких указателей. (Не бывает правил без исключений, и могут возникнуть ситуации, когда поиск нужной позиции вставки в списке будет тоже довольно длительной операцией.)

Для работы методов insert() и erase() требуются итераторы, поэтому мы отложим их подробное рассмотрение на будущее.

Методы reverse(), merge() и unique()

Некоторые методы используются только со списками. Других контейнеров, для которых они были бы определены, просто нет, хотя есть, конечно, алгоритмы, выполняющие практически те же функции. В следующем примере показаны некоторые из таких методов. Программа начинается с заполнения двух целочисленных списков содержимым двух массивов.

Листинг 15.15. Программа LISTPLUS

```
// listplus.cpp
// Демонстрация методов reverse(), merge() и unique()
#include <iostream>
```

продолжение ↗

Листинг 15.15 (*продолжение*)

```
#include <list>
using namespace std:

int main()
   {
   int j:
   list<int> list1, list2:

   int arr1[] = { 40, 30, 20, 10 }:
   int arr2[] = { 15, 20, 25, 30, 35 }:

   for(j=0: j<4: j++)
      list1.push_back( arr1[j] ):   // list1: 40, 30, 20, 10
   for(j=0: j<5: j++)
      list2.push_back( arr2[j] ):   // list2: 15, 20, 25, 30, 35

   list1.reverse():                 // перевернуть list1: 10 20 30 40
   list1.merge(list2):              // объединить list2 с list1
   list1.unique():                  // удалить повторяющиеся элементы 20 и 30

   int size = list1.size():
   while( !list1.empty() )
      {
      cout << list1.front() << ' ': // читать элемент из
                                    //начала
      list1.pop_front():            // вытолкнуть элемент из начала
      }
   cout << endl:
   return 0:
   }
```

Первый список составлен из элементов, расположенных в обратном порядке, поэтому для начала мы его переворачиваем так, чтобы он был отсортирован по возрастанию. После этого действия списки выглядят так:

```
10203040
1520253035
```

Затем выполняется функция merge(). С ее помощью объединяются списки list2 и list1, результат сохраняется в list1. Его новое содержимое:

```
101520202530303540
```

Наконец, мы применяем метод unique() к списку list1. Эта функция находит соседние элементы с одинаковыми значениями и оставляет только один из них. Содержимое списка list1 выводится на экран:

```
10152025303540
```

Для вывода списка используются функции front() и pop_front(). Из них составляется цикл for, таким образом проходится весь список. Каждый элемент по очереди от головы до хвоста выводится на экран, затем выталкивается из списка. Впереди планеты всей после каждого шага оказывается новый элемент. В результате получается, что операция вывода на экран уничтожает список. Может быть, это не всегда то, что нужно, но, тем не менее, это единственный

способ продемонстрировать возможности последовательного доступа к спискам. Итераторы в этом смысле упрощают дело, но к их рассмотрению мы перейдем только в следующем параграфе.

Очереди с двусторонним доступом

Очередь с двусторонним доступом (*deque*) представляет собой нечто похожее и на вектор, и на связный список. Как и вектор, этот тип контейнера поддерживает произвольный доступ (оператор []). Как и к списку, доступ может быть получен к началу и концу очереди. В целом это напоминает вектор с двумя концами. Поддерживаются функции front(), push_front() и pop_front().

Для векторов и двусторонних очередей память резервируется по-разному. Вектор всегда занимает смежные ячейки памяти. Поэтому при его разрастании резервируется новый участок памяти, где контейнер может поместиться целиком. С другой стороны, очередь с двусторонним доступом может размещаться в нескольких сегментах памяти, не обязательно смежных. В этом есть и плюсы, и минусы. Очередь практически всегда будет гарантированно размещена в памяти, но доступ к сегментированным объектам всегда осуществляется с более медленной скоростью. Метод capacity() возвращает наибольшее число элементов вектора, которые можно разместить в памяти без каких-либо махинаций, но он не определен для контейнера deque (очередь с двусторонним доступом), поскольку в данном случае никаких перемещений по памяти не требуется.

Листинг 15.16. Программа DEQUE

```
// deque.cpp
// Демонстрация методов push_back(), push_front(), front()
#include <iostream>
#include <deque>
using namespace std;

int main()
   {
   deque<int> deq;

   deq.push_back(30);          // проталкивание элементов в конец
   deq.push_back(40);
   deq.push_back(50);
   deq.push_front(20);         // проталкивание элементов в начало
   deq.push_front(10);

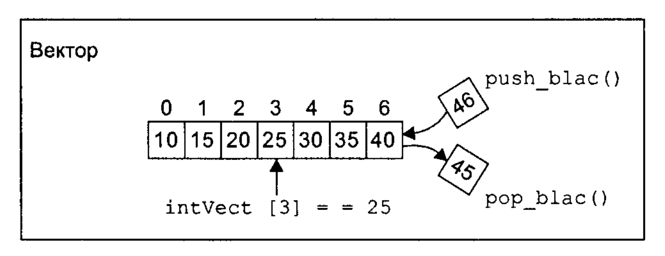
   deq[2] = 33;                // изменение произвольного элемента контейнера

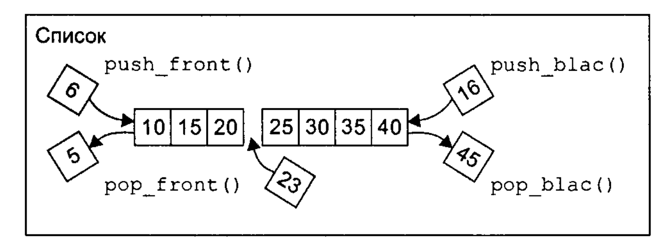
   for(int j=0; j<deq.size(); j++)
      cout << deq[j] << ' ';   // вывести элементы
   cout << endl;
   return 0;
   }
```

Нам уже встречались примеры использования методов push_back(), push_front() и оператора []. К очередям с двусторонним доступом они применяются точно так же, как и к другим контейнерам. Результат работы программы:

```
1020334050
```

На рис. 15.2 показана работа некоторых особо важных методов для трех последовательных контейнеров.





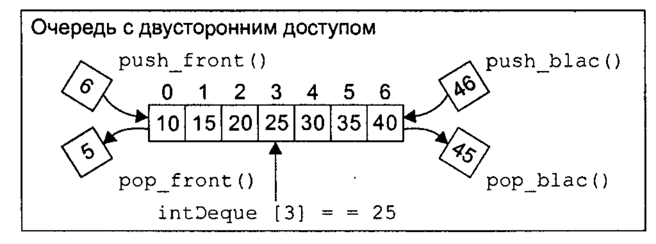


Рис. 15.2. Последовательные контейнеры

Итераторы

Итераторы поначалу могут показаться чем-то несколько мистическим, и все же понятие о них является стержневым для успешного выполнения действий в STL. В этом параграфе мы вначале обсудим двойную роль, которую играют итераторы: роль «интеллектуальных указателей» и мостов, соединяющих алгоритмы с контейнерами. Затем мы покажем некоторые примеры их использования.

Итераторы как интеллектуальные указатели

Часто бывает необходимо выполнить какую-то операцию над всеми элементами контейнера (или каким-то диапазоном данных). Например, это может быть операция вывода содержимого контейнера на экран или суммирования всех элементов. В обычных массивах в C++ для этого используется обращение в цикле ко всем элементам с помощью указателя (или оператора [], впрочем, за ним стоит все тот

же механизм указателей). Например, в нижеследующем отрывке кода производится проход (итерация) по массиву типа float с выводом каждого элемента:

```
float* ptr = start_address;
for(int j=0; j<SIZE; j++)
   cout << *ptr++;
```

Мы снимаем косвенность с указателя ptr с помощью оператора * для получения значения элемента, на который он ссылается, затем инкрементируем ptr, используя ++. После этого значением указателя является адрес следующего элемента контейнера.

Недостатки обычных указателей

Несмотря на приведенный выше пример, использовать указатели с более сложными контейнерами, нежели простые массивы, довольно затруднительно. Во-первых, если элементы контейнера хранятся не последовательно в памяти, а сегментированно, то методы доступа к ним значительно усложняются. Мы не можем в этом случае просто инкрементировать указатель для получения следующего значения. Например, при движении от элемента к элементу в связном списке мы не можем по умолчанию предполагать, что следующий является соседом предыдущего. Приходится идти по цепочке ссылок.

К тому же нам может понадобиться хранить адрес некоторого элемента контейнера в переменной-указателе, чтобы в будущем иметь возможность доступа к нему. Что случится со значением указателя, если мы вставим или удалим что-нибудь из середины контейнера? Он не сможет указывать на корректный адрес, если со структурой данных что-то произошло. Было бы, конечно, здорово, если бы нам не нужно было проверять все наши указатели после каждой вставки и удаления.

Одним из решений проблем такого рода является создание класса «интеллектуальных указателей». Объект такого класса обычно является оболочкой для методов, работающих с обычными указателями. Операторы ++ и * перегружаются и поэтому в курсе того, как нужно работать с элементами контейнера, даже если они расположены не последовательно в памяти или изменяют свое местоположение. Вот как это может выглядеть на практике (приводится только схема работы):

```
class SmartPointer
   {
   private:
      float* p;    //обычный указатель
   public:
      float operator*()
         { }
      float operator++()
         { }
   };
void main()
   {
   . . .
   SmartPointer sptr = start_address;
   for(int j=0; j<SIZE; j++)
      cout << *sptr++;
   }
```

На ком ответственность?

Интересен теперь такой вопрос: должен ли класс интеллектуальных указателей непременно включаться в контейнер, или он может быть отдельным классом? Подход, используемый в STL, заключается как раз в том, чтобы сделать интеллектуальные указатели полностью независимыми и даже дать им собственное имя — *итераторы*. (На самом деле, они представляют собой целое семейство шаблонных классов.) Для создания итераторов необходимо определять их в качестве объектов таких классов.

Итераторы в качестве интерфейса

Кроме того, что итераторы являются «умными указателями» на элементы контейнеров, они играют еще одну немаловажную роль в STL. Они определяют, какие алгоритмы использовать с какими контейнерами. Почему это важно?

Дело в том, что теоретически вы, конечно, можете применить любой алгоритм к любому контейнеру. И, на самом деле, так зачастую можно и нужно делать. Это одно из достоинств STL. Но правил без исключений не бывает. Иногда оказывается, что некоторые алгоритмы ужасно неэффективны при работе с определенными типами контейнеров. Например, алгоритму sort() требуется произвольный доступ к тому контейнеру, который он пытается сортировать. В противном случае приходится проходить все элементы до тех пор, пока не найдется нужный, что, разумеется, займет немало времени, если контейнер солидных размеров. А для эффективной работы алгоритму reverse() требуется иметь возможность обратной и прямой итерации, то есть прохождения контейнера как в обратном, так и в нормальном порядке.

Итераторы предлагают очень элегантный выход из таких неловких положений. Они позволяют определять соответствие алгоритмов контейнерам. Как уже отмечалось, итераторы можно представлять себе в виде кабеля, соединяющего, например, компьютер и принтер. Один конец «вставляется» в контейнер, другой — в алгоритм. Не все кабели можно воткнуть в любой контейнер, и не все кабели можно воткнуть в любой алгоритм. Если вы попытаетесь использовать слишком мощный для данного контейнера алгоритм, то просто вряд ли сможете найти подходящий итератор для их соединения. Попробуйте-попробуйте, а потом посмотрите, как недоволен будет компилятор вашими действиями.

Сколько типов итераторов (кабелей) необходимо для работы? Получается, что всего пять. На рис. 15.3 показаны эти пять категорий в порядке, соответствующем возрастанию их сложности (на начальном уровне «входные» и «выходные» итераторы одинаково просты).

Если алгоритму требуется только лишь продвинуться на один шаг по контейнеру для осуществления последовательного чтения (но не записи!), он может использовать «входной» итератор для связывания себя с контейнером. В реальной практике входные итераторы используются не с контейнерами, а при чтении из файлов или из потока cin.

Если же алгоритму требуется продвинуться вперед на один шаг по контейнеру для осуществления последовательной записи, он может использовать «вы-

ходной» итератор для связывания себя с контейнером. Выходные итераторы используются при записи в файлы или в поток cout.



Рис. 15.3. Категории итераторов

Если алгоритму нужно продвигаться вперед, но при этом совершать как запись, так и чтение, ему придется использовать «прямой» итератор.

В том случае, когда алгоритму требуется продвигаться и вперед, и назад по контейнеру, он использует двунаправленный итератор.

Наконец, если алгоритму нужен незамедлительный доступ к произвольному элементу контейнера безо всяких поэлементных продвижений, используется итератор «произвольного доступа». Он чем-то напоминает массив своей возможностью обращения непосредственно к указанному элементу. Строго определены итераторы, которые могут изменяться с помощью арифметических операций, таких, как

```
iter2 = iter1 + 7;
```

В табл. 15.7 показано, какие операции поддерживаются какими итераторами. Как видите, все итераторы поддерживают оператор ++ для продвижения вперед по контейнеру. Входной итератор может использоваться с оператором * справа от знака равенства (но не слева!):

```
value = *iter;
```

Выходные итераторы могут использоваться с оператором *, но только стоящим слева от знака равенства:

```
*iter = value;
```

Прямой итератор поддерживает и запись, и чтение, а двунаправленный итератор может быть как инкрементирован, так и декрементирован. Итератор про-

извольного доступа поддерживает оператор [] (как и простые арифметические операции + и –) для скоростного обращения к любому элементу.

Таблица 15.7. Возможности итераторов различных типов

Тип итератора	«Шаг вперед» ++	Чтение value=*i	Запись *i=value	«Шаг назад» --	Произвольный доступ [n]
Произвольного доступа	x	x	x	x	x
Двунаправленный	x	x	x	x	
Прямой	x	x	x		
Входной	x		x		
Выходной	x	x			

Алгоритм всегда может использовать итераторы с более широкими возможностями, чем ему требуется. Например, если нужен прямой итератор, совершенно нормальным действием считается использование двунаправленного итератора или итератора с произвольным доступом.

Соответствие алгоритмов контейнерам

Кабель — это не случайная метафора для итераторов, поскольку именно итераторы осуществляют связь между алгоритмами и контейнерами. Сейчас мы сосредоточим внимание на двух концах нашего воображаемого кабеля: на контейнерах и алгоритмах.

Монтаж «кабеля» в контейнерный конец

Если ограничиться рассмотрением только основных контейнеров STL, придется констатировать тот факт, что можно обойтись лишь двумя категориями итераторов. Как показано в табл. 15.8, векторам и очередям с двусторонним доступом вообще все равно, какой итератор используется, а списки, множества, мультимножества, отображения и мультиотображения воспринимают все, кроме итераторов произвольного доступа.

Таблица 15.8. Типы итераторов, поддерживаемые контейнерами

Тип итератора	Вектор	Список	Deque	Множество	Мультимножество	Отображение	Мультиотображение
Произвольного доступа	x		x				
Двунаправленный	x	x	x	x	x	x	x
Прямой	x	x	x	x	x	x	x
Входной	x	x	x	x	x	x	x
Выходной	x	x	x	x	x	x	x

Теперь разберемся, как же STL подключает правильный итератор к контейнеру? При определении итератора нужно указывать, для какого типа контейне-

ра его следует использовать. Например, если вы определили список значений типа int:

```
list<int> iList:          //список int
```

тогда для того, чтобы определить итератор для него, нужно написать:

```
list<int>::iterator iter: //итератор для целочисленного списка
```

После этого STL автоматически делает итератор двунаправленным, так как именно такой тип требуется контейнеру типа список. Итератор для вектора или очереди с двусторонним доступом автоматически делается итератором произвольного доступа.

Такая автоматизация процесса достигается за счет того, что класс итератора конкретного класса является наследником класса более общего итератора. Так и выходит, что итераторы для векторов и очередей с двусторонним доступом являются наследниками класса random_access_iterator, а итераторы для списков — наследниками класса bidirectional_iterator.

Сейчас мы рассмотрим, как к контейнерам подсоединяются «концы» наших «кабелей»-итераторов. Кабель, на самом деле, не вставляется в контейнер. Фигурально выражаясь, он намертво прикручен к нему, как шнур к утюгу. Векторы и очереди с двусторонним доступом прикручены к итераторам произвольного доступа, а списки (и другие ассоциативные контейнеры, о которых мы еще будем говорить несколько позднее) всегда прикручены к двунаправленным итераторам.

Монтаж «кабеля» в алгоритм

Теперь мы посмотрим, что происходит по другую сторону нашей воображаемой проволоки, соединяющей алгоритмы и контейнеры. Итак, что творится со стороны алгоритмов? Разумеется, каждому алгоритму, в зависимости от того, чем он занимается, требуется свой тип итератора. Если нужно обращаться к произвольным элементам контейнера, потребуется итератор произвольного доступа. Если же нужно черепашьим шагом передвигаться от элемента к элементу, то подойдет менее мощный прямой итератор. В табл. 15.9 показаны примеры алгоритмов и требующихся им итераторов.

Таблица 15.9. Типы итераторов, требующиеся представленным алгоритмам

Алгоритм	Входной	Выходной	Прямой	Двунаправленный	Произвольного доступа
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

Опять-таки, несмотря на то что каждому алгоритму требуется определенный уровень возможностей, более мощные итераторы тоже будут работать нормально. Алгоритму replace() требуется прямой итератор, но он будет работать и с двунаправленным итератором, и с итератором произвольного доступа.

Теперь представим, что из разъемов кабеля торчат ножки, как на силовом кабеле компьютера. Это, кстати, показано на рис. 15.4. Те кабели, которым требуются итераторы произвольного доступа, будут иметь по 5 ножек, те, которым требуются двунаправленные итераторы, — 4 ножки. Наконец, те, которым требуются прямые итераторы, будут иметь 3 ножки и т. д.

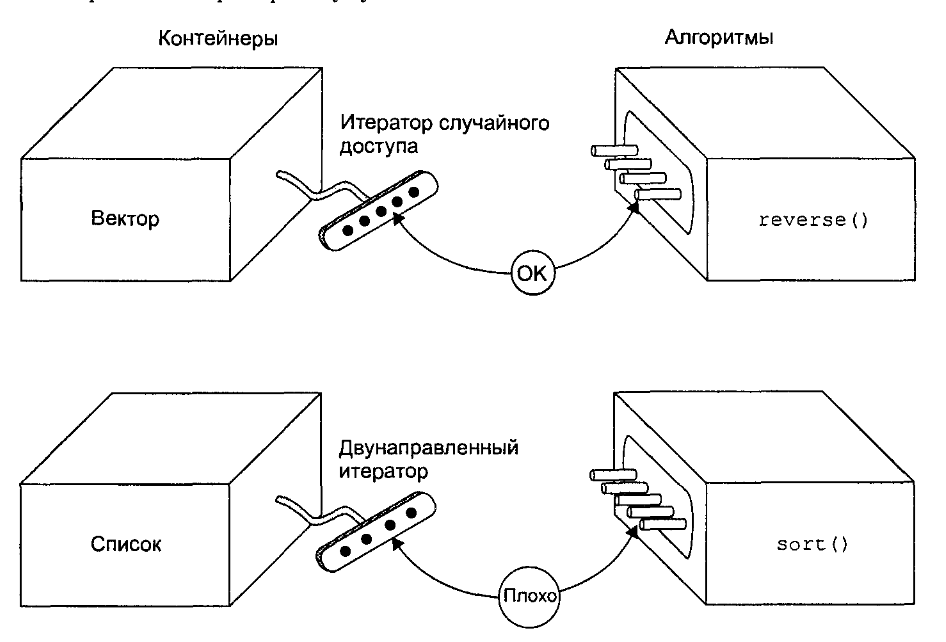


Рис. 15.4. Итераторы, соединяющие контейнеры и алгоритмы

Соответственно, алгоритмический конец «кабеля» имеет некоторое число отверстий под ножки. 5-пиновый кабель можно вставить и в 5-пиновый алгоритм, и в 4-пиновый и т. д. Наоборот — не получится. То есть, переводя на язык терминов STL, нельзя использовать двунаправленные итераторы с алгоритмами, которым требуется произвольный доступ. Таким образом, векторы и очереди с двусторонним доступом, использующие итераторы произвольного доступа, могут сочетаться законным браком с любым алгоритмом, а списки и другие ассоциативные контейнеры с двунаправленным итератором — только с менее мощными алгоритмами.

О чем рассказывают таблицы

Из приведенных выше табл. 15.8 и 15.9 можно узнать, с какими алгоритмами будет работать тот или иной контейнер. Например, из табл. 15.9 видно, что алгоритму sort() требуется итератор произвольного доступа. Таблица 15.8 показыва-

ет, что единственными контейнерами, которые поддерживают такие итераторы, являются векторы и очереди с двусторонним доступом. Бесполезно даже пытаться применить sort() к спискам, множествам, отображениям и т. д.

Любой алгоритм, которому не требуется итератор произвольного доступа, будет работать с любым типом контейнера STL, поскольку все контейнеры используют двунаправленные итераторы, находящиеся лишь на одну ступень ниже уровня произвольного доступа. (Если бы в STL были однонаправленные списки, можно было бы пользоваться прямым итератором, но нельзя было бы применить, например, алгоритм reverse().)

Как видите, относительно малое число алгоритмов реально нуждаются в итераторах произвольного доступа. Поэтому все-таки большинство алгоритмов будет работать с большинством контейнеров.

Перекрытие методов и алгоритмов

Иногда приходится выбирать между методами и алгоритмами с одинаковыми именами. Например, алгоритму find() требуется только входной итератор, поэтому он может использоваться с любым контейнером. Но у множеств и отображений есть свой собственный метод find() (чего нет у последовательных контейнеров). Какую же версию find() следует использовать? В общем случае, если уж имеется метод, дублирующий своим названием алгоритм, так это сделано потому, что алгоритм в данном случае оказывается неэффективным. Поэтому, наверное, при наличии такого выбора следует обратиться к методу.

Работа с итераторами

Использовать итераторы гораздо проще, чем рассказывать о них. Некоторые примеры их применения мы уже видели, когда значения итераторов возвращались методами begin() и end() контейнеров. Мы не обращали внимания на этот факт, полагая, что они работают, как указатели. Теперь посмотрим, как реально используются итераторы с этими и другими функциями.

Доступ к данным

В контейнерах, связанных с итераторами произвольного доступа (векторах и очередях с двусторонним доступом), итерация производится очень просто с помощью оператора []. Такие контейнеры, как списки, которые не поддерживают произвольный доступ, требуют особого подхода. В предыдущих примерах мы использовали «деструктивное чтение» (чтение с разрушением данных) для вывода и одновременного выталкивания элементов итератора. Так было в программах LIST, LISTPLUS. Более практичным действием было бы определение итератора для контейнера. Рассмотрим приведенный ниже пример.

Листинг 15.17. Программа LISTOUT

```
// listout.cpp
// итератор и цикл for для вывода данных
#include <iostream>
#include <list>
```

продолжение ⇨

Листинг 15.17 (*продолжение*)

```
#include <algorithm>
using namespace std:

int main()
   {
   int arr[] = { 2, 4, 6, 8 };
   list<int> theList:

   for(int k=0: k<4: k++)              //заполнить список элементами
      theList.push_back( arr[k] ):     // массива

   list<int>::iterator iter:           //итератор для целочисленного
                                       //списка

   for(iter = theList.begin(): iter != theList.end(): iter++)
      cout << *iter << ' ':            //вывести список
   cout << endl:
   return 0:
   }
```

Программа просто выводит содержимое контейнера theList:

```
2468
```

Мы определяем итератор типа list<int>, соответствующий типу контейнера. Как и в случае с переменной-указателем, до начала использования итератора нужно задать его значение. В цикле for происходит его инициализация значением метода iList.begin(), это указатель на начало контейнера. Итератор может быть инкрементирован оператором ++ для прохождения по элементам, а оператором * можно снять с него косвенность для получения значения того элемента, на который он указывает. Можно даже сравнивать его с чем-нибудь с помощью оператора != и выходить из цикла при достижении итератором конца контейнера (aList.end()).

Вот что будет, если использовать цикл while вместо for:

```
iter = iLst.begin():
while( iter != iList.end() )
   cout << *iter++ << ? ?:
```

Синтаксис *iter++ такой же, как если бы это был указатель, а не итератор.

Вставка данных

Похожий код можно использовать для размещения данных среди существующих элементов контейнера, как показано в следующем примере:

Листинг 15.18. Заполнение контейнера данными

```
// listfill.cpp
// Итератор используется для заполнения контейнера данными
#include <iostream>
#include <list>
using namespace std:

int main()
```

```
{
list<int> iList(5);          // пустой список для хранения 5
                             //значений типа int
list<int>::iterator it;      // итератор
int data = 0;
                             // заполнение списка данными
for(it = iList.begin(); it != iList.end(); it++)
   *it = data += 2;
                             // вывод списка
for(it = iList.begin(); it != iList.end(); it++)
   cout << *it << ' ';
cout << endl;
return 0;
}
```

Первый цикл заполняет контейнер целыми значениями 2, 4, 6, 8, 10, демонстрируя, что перегружаемая операция * может стоять и слева, и справа от знака равенства. Второй цикл предназначен для вывода этих значений.

Алгоритмы и итераторы

Как уже говорилось, алгоритмы используют итераторы в качестве параметров (иногда еще в виде возвращаемых значений). Пример ITERFIND показывает применение алгоритма find() к списку (мы знаем, что это возможно, потому что данному алгоритму требуется только входной итератор).

Листинг 15.19. Программа ITERFIND

```
// iterfind.cpp
// find() возвращает итератор списка
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   list<int> theList(5);        // пустой список для 5 значений int
   list<int>::iterator iter;    // итератор
   int data = 0;
                                // заполнение списка данными
   for(iter = theList.begin(); iter != theList.end(); iter++)
      *iter = data += 2;        //2, 4, 6, 8, 10
                                // поиск числа 8
   iter = find(theList.begin(), theList.end(), 8);
   if( iter != theList.end() )
      cout << "\nНайдено число 8.\n";
   else
      cout << "\nЧисло 8 не найдено.\n";
   return 0;
   }
```

Алгоритм find() имеет три параметра. Первые два — это значения итераторов, определяющие диапазон элементов, по которым может производиться поиск. Третий — искомое значение. Контейнер заполняется теми же значениями 2, 4, 6, 8, 10, что и в предыдущем примере. Затем используется алгоритм find() для

нахождения числа 8. Если он возвращает значение iList.end(), мы понимаем, что достигнут конец контейнера, а искомый элемент не найден. В противном случае должно вернуться значение 8, это означает, что элемент найден. Тогда на экран выводится надпись:

```
Найдено число 8.
```

Можно ли с помощью значения итератора выяснить, где именно в контейнере расположено число 8? Видимо, да. Можно найти смещение относительно начала (iter–iList.begin()). Однако это не является легальным использованием итератора для списков, поскольку итератор должен быть двунаправленным, а значит, над ним нельзя выполнять никакие арифметические операции. Так можно делать с итераторами случайного доступа, например используемыми с векторами и очередями. То есть, если вы ищете значение в векторе v, а не в списке iList, перепишите последнюю часть ITERFIND:

```
iter = find(v.begin(), v.end(), 8);
if( iter != v.end() )
   cout << "\nЧисло 8 расположено по смещению " << (iter - v.begin() );
else
   cout << "\nЧисло 8 не найдено.\n";
```

В результате будет выведено сообщение:

```
Число 8 расположено по смещению 3
```

Вот еще пример, в котором алгоритм использует итераторы в качестве аргументов. Здесь алгоритм copy() используется с вектором. Пользователь определяет диапазон размещений, которые должны быть скопированы из одного вектора в другой. Программа осуществляет это копирование. Диапазон вычисляется с помощью итераторов.

Листинг 15.20. Программа ITERCOPY

```
// itercopy.cpp
// использование итераторов с алгоритмом copy()
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
   {
   int beginRange, endRange;
   int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
   vector<int> v1(arr, arr+10);  // инициализированный вектор
   vector<int> v2(10);           //неинициализированный вектор
   cout << "Введите диапазон копирования (пример: 2 5): ";
   cin >> beginRange >> endRange;

   vector<int>::iterator iter1 = v1.begin() + beginRange;
   vector<int>::iterator iter2 = v1.begin() + endRange;
   vector<int>::iterator iter3;
                                 // копировать диапазон из v1 в v2
   iter3 = copy( iter1, iter2, v2.begin() );
               // (it3 -> последний скопированный элемент)
```

```
  iter1 = v2.begin();           // итерация по диапазону
  while(iter1 != iter3)         // вывести значения из v2
     cout << *iter1++ << ' ';
  cout << endl;
  return 0;
  }
```

Вот один из примеров взаимодействия с программой:

```
Введите диапазон копирования (пример: 2 5): 3 6
17 19 21
```

Мы не выводим целиком вектор, только скопированный диапазон значений. К счастью, copy() возвращает итератор, указывающий на последний элемент (вернее, на элемент, располагающийся следом за последним) из тех, которые были скопированы в контейнер назначения (v2 в данном случае). Программа использует это значение в цикле while для вывода только нужной части элементов.

Специализированные итераторы

В этом разделе мы рассмотрим два специализированных типа итераторов: адаптеры итераторов, которые могут довольно необычным способом изменять поведение обычных итераторов, и потоковые итераторы, благодаря которым входные и выходные потоки могут вести себя как итераторы.

Адаптеры итераторов

В STL различают три варианта модификации обычного итератора. Это *обратный итератор*, *итератор вставки* и *итератор неинициализированного хранения*. Обратный итератор позволяет проходить контейнер в обратном направлении. Итератор вставки создан для изменения поведения различных алгоритмов, таких, как copy() и merge(), чтобы они именно вставляли данные, а не перезаписывали существующие. Итератор неинициализированного хранения позволяет хранить данные в участке памяти, который еще не инициализирован. Он используется в довольно специфических случаях, поэтому здесь мы рассматривать его применение не будем.

Обратные итераторы

Допустим, вам нужно производить итерацию по контейнеру в обратном порядке, от конца к началу. Вы можете попробовать написать такой текст:

```
list<int>::iterator iter;       //обычный итератор
iter = iList.end();             //начать в конце
while( iter != iList.begin() )  // перейти в начало
   cout << *iter-- << ' ';      // декрементировать итератор
```

К сожалению, это работать не будет (прежде всего, у вас получится некорректный диапазон: от n до 1, а не от n–1 до 0).

Для осуществления реверсного прохода контейнеров используются обратные итераторы. В следующем примере показано применение этого специализированного итератора для вывода содержимого списка в обратном порядке.

Листинг 15.21. Программа ITEREV

```
// iterev.cpp
// демонстрация обратного итератора
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 2, 4, 6, 8, 10 };   // массив типа int
   list<int> theList;

   for(int j=0; j<5; j++)            // перенести содержимое массива
      theList.push_back( arr[j] );   // в список

   list<int>::reverse_iterator revit; // обратный итератор

   revit = theList.rbegin();          // реверсная итерация
   while( revit != theList.rend() )   // по списку
      cout << *revit++ << ' ';        // с выводом на экран
   cout << endl;
   return 0;
   }
```

Результаты работы программы:

```
108642
```

При использовании обратного итератора следует использовать методы rbegin() и rend() (не стоит пытаться использовать их с обычным итератором). Интересно, что прохождение контейнера начинается с конца, при этом вызывается метод rbegin(). К тому же значение итератора нужно инкрементировать по мере продвижения от элемента к элементу! Не пытайтесь декрементировать его; revit-- выдаст совсем не тот результат, который вы ожидаете. При использовании reverse_iterator вы движетесь от rbegin() к rend(), инкрементируя итератор.

Итераторы вставки

Некоторые алгоритмы, например copy(), очень любят перезаписывать существующие данные в контейнере назначения. Например, так происходит в следующей программе. В ней копируется содержимое одной очереди с двусторонним доступом в другую.

Листинг 15.22. Программа COPYDEQ

```
// copydeq.cpp
//демонстрация обычного копирования контейнера deque
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
```

```
{
int arr1[] = { 1, 3, 5, 7, 9 };
int arr2[] = { 2, 4, 6, 8, 10 };
deque<int> d1;
deque<int> d2;

for(int j=0; j<5; j++)          //перенос из массивов в очереди
   {
   d1.push_back( arr1[j] );
   d2.push_back( arr2[j] );
   }                            //копирование из d1 в d2
copy( d1.begin(), d1.end(), d2.begin() );

for(int k=0; k<d2.size(); k++)  // вывод d2
   cout << d2[k] << ' ';
cout << endl;
return 0;
}
```

Вот что в результате стало с очередью:

```
13579
```

Содержимое d2 было перезаписано содержимым d1, поэтому в d2 мы не видим никаких следов прежней роскоши. Часто именно это и требуется по логике работы программы, но иногда было бы лучше, если бы алгоритм copy() не перезаписывал, а вставлял данные в контейнер. Такого поведения алгоритма можно добиться с помощью *итератора вставки*. У него есть три интересных инструмента:

- back_inserter вставляет новые элементы в конец;
- front_inserter вставляет новые элементы в начало;
- inserter вставляет новые элементы в указанное место.

Программа DINSITER показывает, как осуществить вставку в конец контейнера.

Листинг 15.23. Программа DINSITER

```
// dinsiter.cpp
// Очереди и итераторы вставки
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
   {
   int arr1[] = { 1, 3, 5, 7, 9 };  //инициализация d1
   int arr2[] = {2, 4, 6};          //инициализация d2
   deque<int> d1;
   deque<int> d2;

   for(int i=0; i<5; i++)           //перенести данные из массивов в
                                    //очереди с двусторонним доступом
      d1.push_back( arr1[i] );
   for(int j=0; j<3; j++)
      d2.push_back( arr2[j] );
```

продолжение

Листинг 15.23 (*продолжение*)

```
                                    //копировать d1 в конец d2
  copy( d1.begin(), d1.end(), back_inserter(d2) );

  cout << "\nd2: ";                 //вывести d2
  for(int k=0; k<d2.size(); k++)
    cout << d2[k] << ' ';
  cout << endl;
  return 0;
  }
```

Для вставки новых элементов в конец используется метод push_back(). При этом новые элементы из исходного контейнера d1 вставляются в конец d2, следом за существующими элементами. Контейнер d1 остается без изменений. Программа выводит на экран содержимое d2:

```
d2: 2 4 6 1 3 5 7 9
```

Если бы мы указали в программе, что элементы нужно вставлять в начало

```
copy( d1.begin(), d1.end(), front_inserter(d2) );
```

тогда новые элементы оказались бы перед существующими данными d2. Механизм, который стоит за этим, это просто метод push_front(), вставляющий элементы в начало и переворачивающий порядок их следования. В этом случае содержимое d2 было бы таким:

```
9 7 5 3 1 2 4 6
```

Можно вставлять данные и в произвольное место контейнера, используя версию inserter итератора. Например, вставим новые данные в начало d2:

```
copy( d1.begin(), d1.end(), inserter(d2, d2.begin() );
```

Первым параметром inserter является имя контейнера, в который нужно переносить элементы, вторым — итератор, указывающий позицию, начиная с которой должны вставляться данные. Так как inserter использует метод insert(), то порядок следования элементов не изменяется. Результирующий контейнер будет содержать следующие данные:

```
1 3 5 7 9 2 4 6
```

Изменяя второй параметр inserter, мы можем указывать произвольное место контейнера.

Обратите внимание на то, что front_inserter не может использоваться с векторами, потому что у последних отсутствует метод push_front(); доступ возможен только к концу такого контейнера.

Потоковые итераторы

Потоковые итераторы позволяют интерпретировать файлы и устройства ввода/вывода (потоки cin и cout), как итераторы. А значит, можно использовать файлы и устройства ввода/вывода в качестве параметров алгоритмов! (Еще одно подтверждение гибкости применения итераторов для связи алгоритмов и контейнеров.)

Основное предназначение входных и выходных итераторов — как раз-таки поддержка классов потоковых итераторов. С их помощью можно осуществлять применение соответствующих алгоритмов напрямую к потокам ввода/вывода.

Потоковые итераторы — это, на самом деле, объекты шаблонных классов разных типов ввода/вывода.

Существует два потоковых итератора: ostream_iterator и istream_iterator. Рассмотрим их по порядку.

Класс ostream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, который имеет дело с выходным итератором. В следующем небольшом примере мы используем его как параметр copy().

Листинг 15.24. Программа OUTITER

```
// outiter.cpp
// Демонстрация ostream_iterator
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 10, 20, 30, 40, 50 };
   list<int> theList;

   for(int j=0; j<5; j++)                        //перенести массив в список
      theList.push_back( arr[j] );

   ostream_iterator<int> ositer(cout, ", ");     //итератор
                                                 //ostream
   cout << "\nСодержимое списка: ";
   copy(theList.begin(), theList.end(), ositer); //вывод
                                                 //списка
   cout << endl;
   return 0;
   }
```

Мы определяем итератор ostream для чтения значений типа int. Двумя параметрами конструктора являются поток, в который будут записываться значения, и строка, которая будет выводиться вслед за каждым из них. Значением потока обычно является имя файла или cout; в данном случае это cout. При записи в этот поток может использоваться строка-разделитель, состоящая из любых символов. Здесь мы используем запятую и пробел.

Алгоритм copy() копирует содержимое списка в поток cout. Итератор выходного потока используется в качестве его третьего аргумента и является именем объекта назначения.

На экране в результате работы программы мы увидим:

```
Содержимое списка: 10, 20, 30, 40, 50,
```

В следующем примере показано, как использовать этот подход для записи содержимого контейнера в файл.

Листинг 15.25. Программа FOUTITER

```
// foutiter.cpp
// Демонстрация работы ostream_iterator с файлами
#include <fstream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 11, 21, 31, 41, 51 };
   list<int> theList;

   for(int j=0; j<5; j++)          //Передача данных
      theList.push_back( arr[j] ); //из массива в список
   ofstream outfile("ITER.DAT");   //создание файлового объекта

   ostream_iterator<int> ositer(outfile, " "); //итератор
                                   //записать список в файл
   copy(theList.begin(), theList.end(), ositer);
   return 0;
   }
```

Необходимо определить файловый объект класса ofstream и ассоциировать его с конкретным файлом (в программе ассоциируем его с файлом ITER.DAT). При записи в файл желательно использовать удобочитаемые разделители. Здесь мы между элементами вставляем просто символ пробела.

Вывод на экран в этой программе не производится, но с помощью любого текстового редактора можно просмотреть содержимое файла ITER.DAT:

```
11 21 31 41 51
```

Класс istream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, работающего с входным итератором. На примере программы INITER покажем, как такие объекты могут являться сразу двумя аргументами алгоритма copy(). Введенные с клавиатуры (поток cin) числа в формате с плавающей запятой сохраняются в контейнере типа список.

Листинг 15.26. Программа INITER

```
// initer.cpp
// Демонстрация istream_iterator
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
   {
   list<float> fList(5);            // неинициализированный список

   cout << "\nВведите 5 чисел (типа float): ";
                                    // итераторы istream
   istream_iterator<float> cin_iter(cin);  // cin
```

```
istream_iterator<float> end_of_stream;   //eos (конец потока)
                                         // копировать из cin в fList
copy( cin_iter, end_of_stream, fList.begin() );

cout << endl;                            // вывести fList
ostream_iterator<float> ositer(cout, "--");
copy(fList.begin(), fList.end(), ositer);
cout << endl;
return 0;
}
```

Взаимодействие программы с пользователем может выглядеть, например, следующим образом:

```
Введите 5 чисел (типа float): 1.1 2.2 3.3 4.4 5.5
1.1--2.2--3.3--4.4--5.5--
```

Для copy() необходимо указывать и верхний, и нижний предел диапазона значений, поскольку данные, приходящие с cin, являются исходными, а не конечными. Программа начинается с того, что мы соединяем istream_iterator с потоком cin, который определен с помощью конструктора с одним параметром как cin_iter. Но что у нас с концом диапазона? Используемый по умолчанию конструктор класса stream_iterator без аргументов выполняет здесь особую роль. Он всегда создает объект istream_iterator для указания конца потока.

Как пользователь сообщает об окончании ввода данных? Нажатием комбинации клавиш Ctrl+Z, в результате чего в поток передается стандартный символ конца файла. Иногда требуется нажать Ctrl+Z несколько раз. Нажатие Enter не приведет к установке признака окончания файла, хотя и поставит ограничитель чисел.

ostream_iterator используется для вывода содержимого списка. Впрочем, для этих целей можно использовать множество других инструментов.

Любые операции вывода на экран, даже такие, как просто вывод строки «Введите пять чисел в формате float», необходимо выполнять не только до использования итератора istream, но даже до его определения, поскольку с того момента, как он определяется в программе, вывод на экран оказывается недоступен: программа ожидает ввода данных.

В следующем примере вместо потока cin используется входной файл, из которого берутся данные для программы. Пример также демонстрирует работу с алгоритмом copy().

Листинг 15.27. Программа FINITER

```
// finiter.cpp
// Демонстрация работы istream_iterator с файлами
#include <iostream>
#include <list>
#include <fstream>
#include <algorithm>
using namespace std;

int main()
   {
```

продолжение

Листинг 15.27 (*продолжение*)

```
list<int> iList;                 // пустой список
ifstream infile("ITER.DAT");     // создать входной файловый объект
                                 // (файл ITER.DAT должен уже существовать)
                                 // итераторы istream
istream_iterator<int> file_iter(infile); // файл
istream_iterator<int> end_of_stream;     // eos (конец потока)
                                 // копировать данные из входного файла в iList
copy( file_iter, end_of_stream, back_inserter(iList) );

cout << endl;                    // вывести iList
ostream_iterator<int> ositer(cout, "--");
copy(iList.begin(), iList.end(), ositer);
cout << endl;
return 0;
}
```

Результат работы программы:

```
11--21--31--31--41--51--
```

Объект ifstream используется для представления файла ITER.DAT, который к моменту запуска программы должен уже физически существовать на диске и содержать необходимые данные (в программе FOUTITER мы его, если помните, создавали).

Вместо cout, как и в случае с итератором istream в программе INITER, мы пользуемся объектом класса ifstream под названием infile. Для обозначения конца потока используется все тот же объект.

Еще одно изменение, сделанное в этой программе, заключается в том, что мы используем back_inserter для вставки данных в iList. Таким образом, этот контейнер можно сделать изначально пустым, не задавая его размеров. Так имеет смысл делать при чтении входных данных, поскольку заранее неизвестно, сколько элементов будет введено.

Ассоциативные контейнеры

Мы убедились в том, что последовательные контейнеры (векторы, списки и очереди с двусторонним доступом) хранят данные в фиксированной линейной последовательности. Поиск конкретного элемента в таком контейнере (если неизвестен или недоступен его индекс или он не находится в одном из концов контейнера) — это длительная процедура, включающая в себя пошаговый переход от позиции к позиции.

В ассоциативных контейнерах элементы не организуются в последовательности. Они представляют собой более сложные структуры, что дает большой выигрыш в скорости поиска. Обычно ассоциативные контейнеры — это «деревья», хотя возможны и другие варианты, например таблицы. В любом случае, главным преимуществом этой категории контейнеров является скорость поиска.

Давайте поговорим о поиске. Он производится с помощью *ключей*, обычно представляющих собой одно численное или строковое значение, которое может быть как атрибутом объекта в контейнере, так и самим объектом.

Рассмотрим две основные категории ассоциативных контейнеров STL: множества и отображения.

Множество хранит объекты, содержащие ключи. Отображения можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения.

Во множествах, как и в отображениях, может храниться только один экземпляр каждого ключа. А каждому ключу, в свою очередь, соответствует уникальное значение. Такой подход используется в словарях, когда каждому слову ставится в соответствие только одна статья. Но в STL есть способ обойти это ограничение. *Мультимножества* и *мультиотображения* аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют несколько общих методов с другими контейнерами. Тем не менее некоторые алгоритмы, такие, как lower_bound() и equal_range(), характерны только для них. Кроме того, некоторые методы могут быть применены для любых контейнеров, кроме ассоциативных. Среди них семейство методов проталкивания и выталкивания (push_back() и т. п.). Действительно, нет особого смысла в их применении к данной категории контейнеров, поскольку все равно элементы должны проталкиваться и выталкиваться в определенном порядке, но не в конец или начало контейнера.

Множества и мультимножества

Множества часто используются для хранения объектов пользовательских классов. Ниже в этой главе мы приведем примеры применения множеств к объектам класса employee. Но и более тривиальные типы тоже множествами не обижены. На рис. 15.5 показана схема работы с этой категорией контейнеров. Объекты упорядочены, и целый объект является ключом.

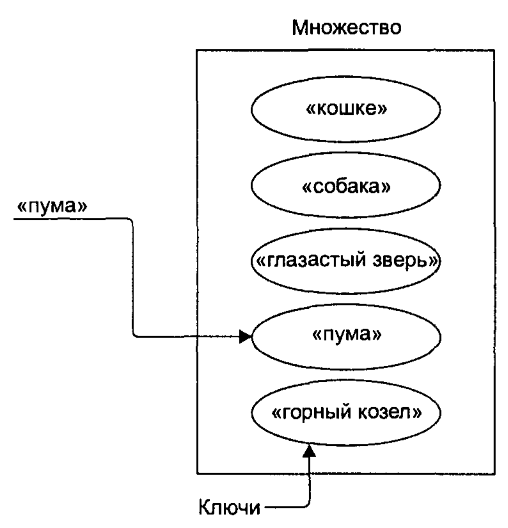


Рис. 15.5. Множество объектов типа string

В нашем первом примере SET мы продемонстрируем множество, содержащее объекты класса string.

Листинг 15.28. Программа SET

```
// set.cpp
// Множество, хранящее обьекты типа string
#pragma warning (disable:4786) //для работы с множествами //(только для компиляторов
Microsoft)
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
   {                              // Массив строковых обьектов
   string names[] = {"Juanita", "Robert",
                     "Mary", "Amanda", "Marie"};
                                  // Инициализировать
                                  //множество массивом
   set<string, less<string> > nameSet(names, names+5);
                                  // итератор для множества
   set<string, less<string> >::iterator iter;

   nameSet.insert("Yvette");      // вставка элементов
   nameSet.insert("Larry");
   nameSet.insert("Robert");      // никакого эффекта: такой
                                  //элемент уже имеется
   nameSet.insert("Barry");
   nameSet.erase("Mary");         // удаление элемента
                                  // вывод размера множества
   cout << "\nРазмер=" << nameSet.size() << endl;
   iter = nameSet.begin();        // вывод элементов множества
   while( iter != nameSet.end() )
      cout << *iter++ << '\n';

   string searchName;             //получение искомого имени от
                                  //пользователя
   cout << "\nВведите искомое имя: ";
   cin >> searchName;
                                  // поиск соответствующего
                                  // запросу имени
   iter = nameSet.find(searchName);
   if( iter == nameSet.end() )
      cout << "Имя " << searchName << " ОТСУТСТВУЕТ во множестве.";
   else
      cout << "Имя " << *iter << " ПРИСУТСТВУЕТ во множестве.";
   cout << endl;
   return 0;
   }
```

Директива

```
#pragma warning (disable:4786)
```

может понадобиться компилятору Microsoft при использовании файлов с множествами или отображениями. Она отключает предупреждение 4786 («в отладочных данных идентификатор был усечен до 255 символов»), появление которого выглядит, как ошибка. Это указание транслятору должно предшествовать

директивам #include всех файлов, которые могут вызвать проблему, а не только указанных. Pragma — это такая специфическая директива компилятора, служащая для настройки выполняемых им операций.

Для определения множества необходимо указать тип хранимых в нем объектов. В нашем примере, например, это объекты класса string. К тому же необходимо указать функциональный объект, который будет использоваться для упорядочивания элементов множества. Здесь мы использовали less<>, применяя его к строковым объектам.

Как видите, множество имеет интерфейс, во многом сходный с другими контейнерами STL. Мы можем инициализировать множество массивом, вставлять данные методом insert(), выводить их с помощью итераторов...

Для нахождения элементов мы используем метод find(). (Последовательные контейнеры имеют одноименный алгоритм.) Вот пример взаимодействия с программой SET. Здесь пользователь вводит в качестве искомого имени «George»:

```
Размер=7
Amanda
Barry
Juanita
Larry
Marie
Robert
Yvette
Введите искомое имя: George
Имя George ОТСУТСТВУЕТ во множестве.
```

Конечно, скорость поиска трудно заметить на примере таких крошечных множеств, однако, поверьте, при достаточно большом количестве элементов ассоциативные контейнеры сильно выигрывают в поиске по сравнению с последовательными.

Рассмотрим одну довольно важную пару методов, которая может использоваться только с ассоциативными контейнерами. А именно, в нашем примере SETRANGE продемонстрировано применение lower_bound()и upper_bound().

Листинг 15.29. Программа SETRANGE

```
// setrange.cpp
// Тестирование работы с диапазонами во множестве.
#pragma warning (disable:4786) //для работы с множествами /
                               //(только для компиляторов Microsoft)
#include <iostream>
#include <set>
#include <string>
using namespace std:

int main()
   {                           // множество обьектов string
   set<string. less<string> > organic:
                               // итератор множества
   set<string. less<string> >::iterator iter:

   organic.insert("Curine"):   // вставка компонентов класса
                               //organic
```

продолжение ⇗

Листинг 15.29 (*продолжение*)

```
organic.insert("Xanthine");
organic.insert("Curarine");
organic.insert("Melamine");
organic.insert("Cyanimide");
organic.insert("Phenol");
organic.insert("Aphrodine");
organic.insert("Imidazole");
organic.insert("Cinchonine");
organic.insert("Palmitamide");
organic.insert("Cyanimide");

iter = organic.begin();     // вывод множества
while( iter != organic.end() )
   cout << *iter++ << '\n';

string lower, upper;         // вывод значений из диапазона
cout << "\nВведите диапазон (например, C Czz): ";
cin >> lower >> upper;
iter = organic.lower_bound(lower);
while( iter != organic.upper_bound(upper) )
   cout << *iter++ << '\n';
return 0;
}
```

Программа вначале выводит полный список элементов множества класса organic. Затем пользователя просят ввести пару ключевых значений, задающих диапазон внутри множества. Элементы, входящие в этот диапазон, выводятся на экран. Пример работы программы:

```
  Aphrodine
  Cinchonine
  Curarine
  Curine
  Cyanimide
  Imidazole
  Melamine
  Palmitamide
  Phenol
  Xanthine

Введите диапазон (например, C Czz): Aaa Curb
  Aphrodine
  Cinchonine
  Curarine
```

Метод lower_bound() берет в качестве аргумента значение того же типа, что и ключ. Он возвращает итератор, указывающий на первую запись множества, значение которой не меньше аргумента (что значит «не меньше», в каждом конкретном случае определяется конкретным функциональным объектом, используемым при определении множества). Функция upper_bound() возвращает итератор, указывающий на элемент, значение которого больше, чем аргумент. Обе эти функции позволяют, как мы и показали в нашем примере, задавать диапазон значений в контейнере.

Отображения и мультиотображения

В *отображении* всегда хранятся пары значений, одно из которых представляет собой ключевой объект, а другое — объект, содержащий значение. Ключевой объект содержит, естественно, ключ, по которому ищется значение. Объект-значение содержит некие данные, которые обычно и интересуют пользователя, запросившего что-то с помощью ключа. В ключевом объекте могут храниться строки, числа или более сложные объекты. Значениями зачастую являются тоже строки, числа или другие объекты, вплоть до контейнеров!

Например, ключом может быть слово, а значением — число, говорящее о том, сколько раз это слово встречалось в тексте. Такое отображение задает *частотную таблицу*. Или, например, ключом может быть слово, а значением — список номеров страниц. Такое решение может быть употреблено для создания индекса, как в конце этой книги. На рис. 15.6 показана ситуация, в которой ключами являются слова, а значениями — определения, как в обыкновенном словаре.

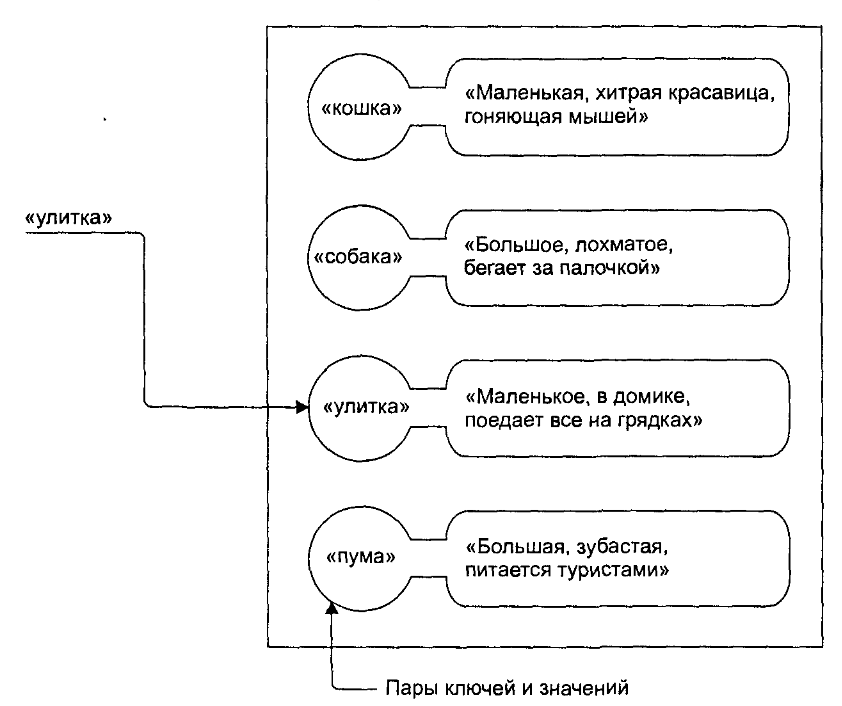


Рис. 15.6. Отображение «слово — фраза»

Одним из самых популярных примеров использования отображений являются ассоциативные массивы. В обыкновенных массивах индекс всегда является целым числом. С его помощью, как известно, осуществляется доступ к конкретным элементам. Так, в выражении anArray[3] число 3 является индексом. Несколько иначе дело обстоит в ассоциативных массивах. Тип данных индекса массива можно задавать самостоятельно. При этом появляется уникальная возможность написать, например, такое выражение: anArray[?jane?].

Ассоциативный массив

Давайте рассмотрим небольшой пример отображения, используемого в качестве ассоциативного массива. Ключами будут названия штатов, а значениями — их население.

Листинг 15.30. Программа ASSO_ARR

```
// asso_arr.cpp
// Демонстрация отображения, используемого в качестве //ассоциативного массива
#pragma warning (disable:4786)  //для отображений
//(только компиляторы Microsoft)
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
   {
   string name;
   int pop;

   string states[] = { "Wyoming", "Colorado", "Nevada",
                       "Montana", "Arizona", "Idaho"};
   int pops[] = { 470, 2890, 800, 787, 2718, 944 };

   map<string, int, less<string> > mapStates;      //отображение
   map<string, int, less<string> >::iterator iter; //итератор

   for(int j=0; j<6; j++)
      {
      name = states[j];                //получение данных из массивов
      pop = pops[j];
      mapStates[name] = pop;           //занесение их в отображение
      }
   cout << "Введите название штата: "; //получение имени штата
   cin >> name;
   pop = mapStates[name];              //найти население штата
   cout << "Население: " << pop << " 000\n";

   cout << endl;                       //вывод всего отображения
   for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
      cout << (*iter).first << ' ' << (*iter).second << "000\n";
   return 0;
   }
```

При запуске программы у пользователя запрашивается название штата. Полученное значение используется в качестве ключа для поиска в отображении значения населения и вывода его на экран. После этого выводится все содержимое отображения: все пары штат — население. Приведем пример работы программы:

```
Arizona 2718 000
Colorado 2890 000
Idaho 944 000
```

```
Montana 787 000
Nevada 800 000
Wyoming 470 000
```

При использовании множеств и отображений к скорости поиска никаких претензий не возникает. Вот и в данном случае программа очень быстро находит значение населения по введенному названию штата (а если в отображении хранятся миллионы пар, представляете, каким эффективным оказывается этот подход!). Ну, всегда приходится чем-то жертвовать, и итерация в данном типе контейнеров не такая быстрая, как в последовательных контейнерах. Обратите внимание, имена штатов расположены в алфавитном порядке, хотя были заданы в массиве хаотическим образом.

Определение отображения имеет три шаблонных аргумента:

```
map<string, int, less<string> > maStates;
```

Первый из них задает тип ключа. В данном случае это string, потому что в строке задается имя штата. Второй аргумент определяет тип значений, хранящихся в контейнере (int в нашем примере — это население в тысячах человек). Третий аргумент задает порядок сортировки ключей. Мы здесь сортируем их по алфавиту, для чего используем less<string>. Еще для этого отображения задается итератор.

Входные данные программы изначально хранятся в двух разных массивах (в реальных проектах, возможно, это будут специальные файлы). Для занесения их в контейнер мы пишем:

```
mapStates[name] = pop;
```

Это элегантное выражение сильно напоминает вставку в обыкновенный массив, однако индексом такого «массива» является строка name, а не просто какое-то целое число.

После того как пользователь вводит названия штата, программа ищет соответствующее значение населения:

```
pop = mapStates[name];
```

Кроме того, что можно использовать синтаксис обращения к элементам по индексам, напоминающий обращение к массивам, можно получить доступ одновременно к обеим частям отображения: ключам и значениям. Делается это при помощи итераторов. Ключ получают по (*iter).first, а значение — по (*iter).second. При других вариантах обращения итератор работает так же, как в других контейнерах.

Хранение пользовательских объектов

До этого момента в наших примерах мы хранили объекты базовых типов. Между тем, одно из достоинств STL заключается как раз в том, что объекты пользовательских типов тоже являются полноправными данными, которые можно хранить, над которыми можно выполнять все те же операции. В этом разделе мы обратимся к теме работы с такими типами.

Множество объектов person

Начнем наш разговор с класса person, в котором содержатся фамилии, имена и телефоны людей. Мы будем создавать компоненты этого класса и вставлять их во множество, заполняя тем самым виртуальную телефонную книгу. Пользователь взаимодействует с программой, вводя имя человека. На экран выводится результат поиска данных, соответствующих указанному имени. Применим для решения этой задачи мультимножество, чтобы два и более объекта person могли иметь одинаковые имена. Приведем листинг программы SETPERS.

Листинг 15.31. Программа SETPERS

```
// setpers.cpp
// Применение мультимножества для хранения объектов person
#pragma warning (disable:4786)     // для множеств (для
                                   // компиляторов фирмы Microsoft)
#include <iostream>
#include <set>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:                         // конструктор по умолчанию
      person() : lastName("пусто").
                 firstName("пусто"), phoneNumber(0)
         { }
                                   // конструктор с тремя параметрами
      person(string lana, string fina, long pho) :
               lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);

      void display() const         // вывод данных о людях
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tТелефон: " << phoneNumber;
         }
   };
                                   // оператор < для класса person
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
                                   // оператор == для класса person
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
```

```
////////////////////////////////////////////////////////////////
int main()
  {                              // создание обьектов person
  person pers1("Deauville", "William", 8435150);
  person pers2("McDonald", "Stacey", 3327563);
  person pers3("Bartoski", "Peter", 6946473);
  person pers4("KuangThu", "Bruce", 4157300);
  person pers5("Wellington", "John", 9207404);
  person pers6("McDonald", "Amanda", 8435150);
  person pers7("Fredericks", "Roger", 7049982);
  person pers8("McDonald", "Stacey", 7764987);
                                 // мультимножество класса person
  multiset< person, less<person> > persSet;
                                 // итератор этого мультимножества
  multiset<person, less<person> >::iterator iter;

  persSet.insert(pers1);         // занести обьекты person в
                                 //мультимножество
  persSet.insert(pers2);
  persSet.insert(pers3);
  persSet.insert(pers4);
  persSet.insert(pers5);
  persSet.insert(pers6);
  persSet.insert(pers7);
  persSet.insert(pers8);

  cout << "\nЧисло записей: " << persSet.size();

  iter = persSet.begin();        //Вывод содержимого
                                 //  мультимножества
  while( iter != persSet.end() )
     (*iter++).display();
                                 // получение имени и фамилии
  string searchLastName, searchFirstName;
  cout << "\n\nВведите фамилию искомого человека: ";
  cin >> searchLastName;
  cout << "Введите имя: ";
  cin >> searchFirstName;
       // создание обьекта с заданными значениями атрибутов
  person searchPerson(searchLastName, searchFirstName, 0);
       // сосчитать количество людей с таким именем
  int cntPersons = persSet.count(searchPerson);
  cout << "Число людей с таким именем: " << cntPersons;

       // вывести все записи, отвечающие запросу
  iter = persSet.lower_bound(searchPerson);
  while( iter != persSet.upper_bound(searchPerson) )
     (*iter++).display();
  cout << endl;
  return 0;
  }    // end main()
```

Необходимые методы

Для работы с контейнерами STL классу person требуется несколько общих методов. Они представляют собой конструкторы по умолчанию (без аргументов), которые в данном примере не очень нужны, но вообще-то обычно бывают довольно

полезны. Кроме того, весьма и весьма полезными оказываются перегружаемые операции < и ==. Все эти методы используются списковым классом и различными алгоритмами. В других ситуациях могут понадобиться какие-то другие специфические методы. (Как и при работе с большинством классов, наверное, нелишним будет иметь под рукой перегружаемую операцию присваивания, конструктор копирования и деструктор, но сейчас мы не будем заводить разговор об этом, дабы не загромождать листинг.)

Перегружаемые операции < и == должны иметь константные параметры. Вообще говоря, даже лучше сделать их дружественными, но и в виде обычных методов они вполне подходят для работы.

Порядок сортировки

Перегружаемая операция < задает метод сортировки элементов множества. В нашей программе SETPERS мы определяем его таким образом, чтобы сортировались фамилии, а в случае их совпадения — еще и имена.

Ниже приводится некий пример взаимодействия пользователя с программой. Вначале на экран выводится весь список хранимых значений (конечно, так не следует делать в случае, если вы имеете дело с реальной базой данной с большим числом элементов). Поскольку данные хранятся в мультимножестве, они сортируются автоматически. После этого у пользователя запрашивается имя человека, он вводит «McDonald», затем «Stacey» (вначале вводится фамилия). А в нашем списке имеется два полных тезки, и на экран выводится информация об обоих.

```
Число записей: 8
Bartoski.Peterтелефон: 6946473
Deauville.Williamтелефон: 8435150
Fredericks.Rogerтелефон: 7049982
KuangThu.Bruceтелефон: 4157300
McDonald.Amandaтелефон: 8435150
McDonald.Staceyтелефон: 3327563
McDonald.Staceyтелефон: 7764987
Wellington.Johnтелефон: 9207404

Введите фамилию искомого человека: McDonald
Введите имя: Stacey
Число людей с таким именем: 2
McDonald.Staceyтелефон: 3327563
McDonald.Staceyтелефон: 7764987
```

Схожесть с базовыми типами

Как видите, едва мы определили класс, как получили возможность работы контейнера с ним теми же способами, что и с обычными переменными базовых типов.

Метод size() используется для вывода количества записей. Затем производится итерация по контейнеру с целью вывода на экран всех элементов.

Поскольку мы имеем дело с мультимножеством, методы lower_bound() и upper_bound() дают возможность выводить все элементы, входящие в указанный диапазон. В нашем примере верхняя граница совпадает с нижней, поэтому просто выводятся все записи, содержащие данные о людях с заданным именем. Обратите внимание: мы создаем фиктивную запись с данными о человеке (или о несколь-

ких людях) с тем же именем (именами), которое пользователь указал в запросе. При этом значения функций lower_bound() и upper_bound() должны соответствовать именно указанным записям, в отличие от ситуации, описанной выше (в самом списке их значения совпадали).

Список объектов класса person

Поиск человека с заданным именем оказывается очень быстрым при работе с множествами или мультимножествами. Если же нам важней не быстрый поиск, а быстрая вставка или удаление объектов типа person, то лучше обратиться к спискам. В следующем примере показано, как именно применять списки объектов пользовательских классов на практике.

Листинг 15.32. Программа LISTPERS

```
// listpers.cpp
// Использование списка для хранения обьектов person
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :              // конструктор без аргументов
          lastName("пусто"), firstName("пусто"), phoneNumber(0L)
         { }
                              // конструктор с тремя аргументами
      person(string lana, string fina, long pho) :
            lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);
      friend bool operator!=(const person&, const person&);
      friend bool operator>(const person&, const person&);

      void display() const    // вывод всех данных
         {
         cout << endl << lastName << ",\t" << firstName
            << "\t\tТелефон: " << phoneNumber;
         }

      long get_phone() const // вернуть номер телефона
         { return phoneNumber; }
   };
                              // перегруженный == для класса person
bool operator==(const person& p1, const person& p2)
   {
```

продолжение

Листинг 15.32 (*продолжение*)

```
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
                              // перегруженный < для класса person
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
                              // перегруженный != для класса person
bool operator!=(const person& p1, const person& p2)
   { return !(p1==p2); }
                              // перегруженный > для класса person
bool operator>(const person& p1, const person& p2)
   { return !(p1<p2) && !(p1==p2); }
///////////////////////////////////////////////////////////
int main()
   {
   list<person> persList;     // список объектов типа person
                              // итератор для этого списка
   list<person>::iterator iter1;
                              // занести объекты в список
   persList.push_back( person("Deauville", "William", 8435150) );
   persList.push_back( person("McDonald", "Stacey", 3327563) );
   persList.push_back( person("Bartoski", "Peter", 6946473) );
   persList.push_back( person("KuangThu", "Bruce", 4157300) );
   persList.push_back( person("Wellington", "John", 9207404) );
   persList.push_back( person("McDonald", "Amanda", 8435150) );
   persList.push_back( person("Fredericks", "Roger", 7049982) );
   persList.push_back( person("McDonald", "Stacey", 7764987) );

   cout << "\nЧисло записей: " << persList.size();

   iter1 = persList.begin(); // вывод содержимого списка
   while( iter1 != persList.end() )
      (*iter1++).display();

                              // найти информацию о людях с именами
                              // и фамилиями, указанными в запросе
   string searchLastName, searchFirstName;
   cout << "\n\nВведите фамилию искомого человека: ";
   cin >> searchLastName;
   cout << "Введите имя: ";
   cin >> searchFirstName;
                              //создать персону с таким именем
   person searchPerson(searchLastName, searchFirstName, 0L);
      //поиск по листу первого вхождения искомого значения
   iter1 = find(persList.begin(), persList.end(), searchPerson);
   if( iter1 != persList.end() ) //поиск следующих совпадений
      {
      cout << "Есть такой человек (такие люди) в списке: ";
      do
         {
         (*iter1).display(); //вывод найденных записей
         ++iter1;            //продолжение поиска со следующей записи
         iter1 = find(iter1, persList.end(), searchPerson);
```

```
        } while( iterl != persList.end() );
     }
  else
     cout << "Человек с таким именем отсутствует в списке.";

                              // найти человека по номеру телефона
  cout << "\n\nВведите номер телефона (формат 1234567): ";
  long sNumber;               //получить искомый номер
  cin >> sNumber;
                              //итерация по списку
  bool found_one = false;
  for(iterl=persList.begin(); iterl != persList.end(); ++iterl)
     {
     if( sNumber == (*iterl).get_phone() )  //сравнить
                                            //номера
        {
        if( !found_one )
           {
           cout << "Есть человек (или несколько) с таким номером телефона.";
           found_one = true;
           }
        (*iterl).display(); //display the match
        }
     } //end for
  if( !found_one )
     cout << "Человек с таким номером телефона отсутствует в списке";
  cout << endl;
  return 0;
  } //end main()
```

Поиск всех людей с указанным именем

В такой ситуации у нас нет возможности использовать методы lower_bound() и upper_bound(), поскольку в данном случае мы имеем дело и не с множеством, и не с отображением. В списках такие методы отсутствуют. Вместо них мы пользуемся уже знакомой функцией find(). Если функция возвращает значение найденного элемента, нужно снова вызвать ее для осуществления поиска, начиная со следующей записи после найденной. Это несколько усложняет программу, так как приходится вносить в нее цикл с двумя вызовами find().

Поиск всех людей с указанным номером телефона

Сложнее обстоит дело с поиском данных при заданном номере телефона, так как методы этого класса, в том числе find(), предназначены для поиска первичной характеристики. Поэтому приходится «вручную» искать номер телефона, производя итерацию по списку и сравнивая заданный номер телефона с имеющимся в очередной текущей записи:

```
if( sNumber == (*iterl).getphone() )
   . . .
```

Для начала программа просто выводит все записи из списка. Затем запрашивает у пользователя имя и находит соответствующие данные о человеке (или нескольких людях). После этого пользователя просят ввести номер телефона, и программа снова ищет подходящие данные.

Один из примеров работы:

```
Число записей: 8
Deauville.Williamтелефон: 8435150
McDonald.Staceyтелефон: 3327563
Bartoski.Peterтелефон: 6946473
KuangThu.Bruceтелефон: 4157300
Wellington.Johnтелефон: 9207404
McDonald.Amandaтелефон: 8435150
Fredericks.Rogerтелефон: 7049982
McDonald.Staceyтелефон: 7764987

Введите фамилию искомого человека: Wellington
Введите имя: John
Есть такой человек (такие люди) в списке:
Wellington.Johnтелефон: 9207404
Введите номер телефона (формат 1234567): 8435150
Есть человек (или несколько) с таким номером телефона.
Deauville.Williamтелефон: 8435150
McDonald.Amandaтелефон: 8435150
```

Как видите, программа нашла одного человека по указанному имени и двух человек по указанному телефону.

При использовании списков для хранения объектов, нужно объявлять четыре оператора сравнения для конкретного класса: ==, !=, < и >. В зависимости от того, какие алгоритмы используются в программе, определяют те или иные из них, ведь далеко не всегда нужны все операторы сразу. Так, в нашем примере мы могли бы ограничиться лишь определением оператора ==, хотя для полноты картины определили все. А если бы, например, в программе встречался алгоритм sort() для списка, была бы необходима перегружаемая операция <.

Функциональные объекты

Это понятие широко используется в STL. Одним из популярнейших применений функциональных объектов является передача их в качестве аргументов алгоритмам. Тем самым можно регулировать поведение последних. Мы уже упоминали о функциональных объектах в этой главе и даже использовали один из них в программе SORTEMP. Там мы показывали пример предопределенного функционального объекта greater<>(), используемого для сортировки данных в обратном порядке. В этом параграфе будут представлены другие предопределенные функциональные объекты; кроме того, вы узнаете, как написать свой, который, возможно, предоставит даже больший контроль над алгоритмами STL.

Давайте вспомним, что такое функциональный объект. Это просто-напросто функция, которая таким образом пристраивается к классу, что выглядит совсем как обычный объект. Тем не менее в таком классе не может быть компонентных данных, а есть только один метод: перегружаемая операция (). Класс часто делают шаблонным, чтобы можно было работать с разными типами данных.

Предопределенные функциональные объекты

Предопределенные функциональные объекты расположены в заголовочном файле FUNCTIONAL и показаны в табл. 15.10. Там находятся объекты, которые могут работать с большинством операторов C++. В таблице символом T обозначен любой класс: пользовательский или одного из базовых типов. Переменные x и y — объекты класса T, передаваемые в функцию в виде параметров.

Таблица 15.10. Предопределенные функциональные объекты

Функциональный объект	Возвращаемое значение
T = plus(T, T)	x+y
T = minus(T, T)	x–y
T = times(T, T)	x*y
T = divide(T, T)	x/y
T = modulus(T, T)	x%y
T = negate(T)	–x
bool = equal_to(T, T)	x == y
bool = not_equal_to(T, T)	x != y
bool = greater(T, T)	x > y
bool = less(T, T)	x < y
bool = greater_equal(T, T)	x >= y
bool = less_equal(T, T)	x <= y
bool = logical_and(T, T)	x && y
bool = logical_or(T, T)	x || y
bool = logical_not(T, T)	!x

Как видите, имеются функциональные объекты для выполнения арифметических, логических операций и операций сравнения. Рассмотрим пример, в котором очень ловко используется арифметический функциональный объект. Будем работать с классом airtime, представляющим собой значения времени только в часах и минутах, без секунд. Данные такого типа подходят для составления расписания прибытия и отправления поездов или самолетов. В программе демонстрируется функциональный объект plus<>(), с помощью которого можно работать со значениями объектов airtime в контейнере. Приведем ее листинг.

Листинг 15.33. Программа PLUSAIR

```
//plusair.cpp
//использование алгоритма accumulate() и функционального //объекта plus()
#include <iostream>
#include <list>
#include <numeric>          //для accumulate()
using namespace std;
////////////////////////////////////////////////////////////////
class airtime
   {
   private:
      int hours;            //от 0 до 23
      int minutes;          //от 0 до 59
```

продолжение ➔

Листинг 15.33 (*продолжение*)

```
  public:
                                //конструктор по умолчанию
     airtime() : hours(0), minutes(0)
        {  }
                                //конструктор с двумя аргументами
     airtime(int h, int m) : hours(h), minutes(m)
        {  }
     void display() const   //вывод на экран
        { cout << hours << ':' << minutes; }

     void get()             //ввод данных пользователем
        {
        char dummy;
        cout << "\nВведите время (формат 12:59): ";
        cin >> hours >> dummy >> minutes;
        }
                                //перегружаемая операция +
     airtime operator + (const airtime right) const
        {                       //добавить компоненты
        int temph = hours + right.hours;
        int tempm = minutes + right.minutes;
        if(tempm >= 60)      //проверка наличия переноса
           { temph++; tempm -= 60; }
        return airtime(temph, tempm); //возврат суммы
        }
                                //перегруженный  оператор ==
     bool operator == (const airtime& at2) const
        { return (hours == at2.hours) &&
                 (minutes == at2.minutes); }
                                //перегруженный  оператор <
     bool operator < (const airtime& at2) const
        { return (hours < at2.hours) ||
                 (hours == at2.hours && minutes < at2.minutes); }
                                //перегружаемая операция !=
     bool operator != (const airtime& at2) const
        { return !(*this==at2); }
                                //перегружаемая операция >
     bool operator > (const airtime& at2) const
        { return !(*this<at2) && !(*this==at2); }
  };                            //конец класса airtime
////////////////////////////////////////////////////////////////
int main()
   {
   char answer;
   airtime temp, sum;
   list<airtime> airlist;    //список типа airtime

   do {                         //получить значения от пользователя
      temp.get();
      airlist.push_back(temp);
      cout << "Продолжть (y/n)? ";
      cin >> answer;
      } while (answer != 'n');
                                //суммировать все элементы
   sum = accumulate( airlist.begin(), airlist.end(),
                     airtime(0, 0), plus<airtime>() );
```

```
cout << "\nСумма = ";
sum.display();              //вывод суммы на экран
cout << endl;
return 0;
}
```

Кроме функционального объекта, в этой программе представляется алгоритм accumulate(). Существуют две версии этой функции. Версия с тремя аргументами всегда суммирует (с помощью перегруженного +) значения из заданного диапазона. Если же у алгоритма четыре аргумента, как в нашем примере, то может быть использован любой функциональный объект из показанных в табл. 15.10.

Четыре аргумента accumulate() — это итераторы первого и последнего элемента диапазона, инициализационное значение суммы (обычно 0), а также операция, которую следует применить к элементам. В нашей программе мы выполняем сложение (plus<>()), но можно было бы выполнять и вычитание, и деление, и умножение, и другие операции, задаваемые функциональными объектами. Вот пример взаимодействия с программой:

```
Введите время: (формат 12:59) : 3:45
Продолжить (y/n)? y

Введите время: (формат 12:59) : 5:10
Продолжить (y/n)? y

Введите время. (формат 12:59) : 2:55
Продолжить (y/n)? y

Введите время: (формат 12:59) : 0:55
Продолжить (y/n)? n
Сумма = 12:15
```

Алгоритм accumulate() не только яснее и понятнее, чем итерация по контейнеру, но и его использование дает больший эффект.

Функциональный объект plus<>() нуждается в перегруженном операторе +. Причем он должен быть перегружен для класса airtime. Этот оператор должен быть объявлен как const-функция, так как именно это требуется функциональному объекту plus<>().

Прочие арифметические функциональные объекты работают аналогично. Логические функциональные объекты, такие, как logical_and<>(), следует использовать с объектами тех классов, где такое булево представление значения имеет смысл. Например, в качестве объекта может выступать обычная переменная типа bool.

Создание собственных функциональных объектов

Если ни один из готовых предопределенных функциональных объектов вас не устраивает, можно без особых сложностей написать свой. В нашем следующем примере освещаются две ситуации, в которых именно этот способ оказывается предпочтительным. Одна из них включает в себя применение алгоритма sort(), а другая — алгоритма for_each().

Группу элементов очень удобно сортировать, используя отношения, заданные оператором < класса. Но задумывались ли вы когда-нибудь о том, что может произойти, если задаться целью отсортировать контейнер, содержащий не сами объекты, а только ссылки, указатели на них? Хранение указателей вместо объектов — это грамотное решение, особенно в случае больших объемов информации. Такой подход становится эффективным за счет избежания копирования каждого объекта при помещении его в контейнер. Тем не менее проблема сортировки остается, поскольку объекты будут выстроены по адресам указателей, а не по каким-то собственным их атрибутам.

Чтобы заставить sort() работать в случае с контейнером указателей так, как нам нужно, необходимо иметь отдельный функциональный объект, задающий метод сортировки.

Программа, листинг которой представлен чуть ниже, начинается с того, что задается вектор указателей на объекты класса person. Объекты помещаются в вектор, затем сортируются обычным способом, что, разумеется, приводит к упорядочиванию по адресам указателей, а не по атрибутам самих объектов. Это совсем не то, что нам нужно; кроме того, наша сортировка вообще не вызывает никакого эффекта, так как данные изначально вводились подряд, а значит, адреса указателей тоже следовали в возрастающем порядке. После этого вектор сортируется с учетом возникшей непонятной ситуации. Для этого мы используем собственный функциональный объект comparePersons(). Он упорядочивает элементы, на которые ссылаются указатели, а не сами значения указателей. В результате объекты person упорядочиваются в алфавитном порядке. Мы за это и боролись. Приведем листинг.

Листинг 15.34. Программа SORTPTRS

```
// sortptrs.cpp
// сортировка объектов person, хранимых в виде содержимого //указателей
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :              // конструктор по умолчанию
          lastName("пусто"), firstName("пусто"), phoneNumber(0L)
         { }
                              // конструктор с тремя аргументами
      person(string lana, string fina, long pho) :
            lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);
```

```
    void display() const   // вывод персональных данных
       {
       cout << endl << lastName << ",\t" << firstName
            << "\t\tТелефон: " << phoneNumber;
       }
    long get_phone() const // вернуть телефонный номер
       { return phoneNumber; }
   }; //end class person
//-----------------------------------------------------------
// перегруженный < для класса person
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
//-----------------------------------------------------------
// перегруженный == для класса person
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
//-----------------------------------------------------------
// функциональный объект для сравнивания содержимого
//указателей на объекты person
class comparePersons
   {
   public:
   bool operator() (const person* ptrP1,
                    const person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
   };
//-----------------------------------------------------------
//функциональный объект для вывода персональных данных,
//хранимых в указателях
class displayPerson
   {
   public:
   void operator() (const person* ptrP) const
      { ptrP->display(); }
   };
///////////////////////////////////////////////////////////
int main()
   {                        // вектор указателей на объекты person
   vector<person*> vectPtrsPers;
                            //создание персональных данных
   person* ptrP1 = new person("KuangThu", "Bruce", 4157300);
   person* ptrP2 = new person("Deauville", "William", 8435150);
   person* ptrP3 = new person("Wellington", "John", 9207404);
   person* ptrP4 = new person("Bartoski", "Peter", 6946473);
   person* ptrP5 = new person("Fredericks", "Roger", 7049982);
   person* ptrP6 = new person("McDonald", "Stacey", 7764987);

   vectPtrsPers.push_back(ptrP1); //внесение данных во
                                  //множество
   vectPtrsPers.push_back(ptrP2);
```

продолжение ➪

Листинг 15.34 (*продолжение*)

```
    vectPtrsPers.push_back(ptrP3);
    vectPtrsPers.push_back(ptrP4);
    vectPtrsPers.push_back(ptrP5);
    vectPtrsPers.push_back(ptrP6);

    for_each(vectPtrsPers.begin(), //вывод вектора
            vectPtrsPers.end(), displayPerson() );
                                    //сортировка указателей
   sort( vectPtrsPers.begin(), vectPtrsPers.end() );
   cout << "\n\nУпорядочены указатели:";
   for_each(vectPtrsPers.begin(), //вывод вектора
            vectPtrsPers.end(), displayPerson() );

   sort( vectPtrsPers.begin(),    //сортировка реальных данных
         vectPtrsPers.end(), comparePersons() );
   cout << "\n\nУпорядочены персональные данные:";
   for_each(vectPtrsPers.begin(),  //вывод вектора
            vectPtrsPers.end(), displayPerson() );
   while( !vectPtrsPers.empty() )
      {
      delete vectPtrsPers.back(); //удаление персоны
      vectPtrsPers.pop_back();    //выталкивание указателя
      }
   cout << endl;
   return 0;
   } // конец main()
```

Вот как выглядит на экране результат работы программы:

```
KuangThu.Bruceтелефон: 4157300
Deauville.Williamтелефон: 8435150
Wellington.Johnтелефон: 9207404
Bartoski.Peterтелефон: 6946473
Fredericks.Rogerтелефон: 7049982
McDonald.Staceyтелефон: 3327563

Упорядочены указатели:
KuangThu.Bruceтелефон: 4157300
Deauville.Williamтелефон: 8435150
Wellington.Johnтелефон: 9207404
Bartoski.Peterтелефон: 6946473
Fredericks.Rogerтелефон: 7049982
McDonald.Staceyтелефон: 3327563

Упорядочены персональные данные:
Bartoski.Peterтелефон: 6946473
Deauville.Williamтелефон: 8435150
Fredericks.Rogerтелефон: 7049982
KuangThu.Bruceтелефон: 4157300
McDonald.Staceyтелефон: 3327563
Wellington.Johnтелефон: 9207404
```

Вначале выводятся данные в исходном порядке, затем — некорректно отсортированные по указателям, наконец, правильно упорядоченные по именам людей.

Функциональный объект comparePersons()

Если мы возьмем версию алгоритма sort(), имеющую два аргумента

```
sort( vectPtrsPers.begin(), vectPtrsPers.end() );
```

то будут сортироваться только указатели по их адресам в памяти. Такая сортировка требуется крайне редко. Чтобы упорядочить объекты person по именам, мы воспользуемся алгоритмом sort() с тремя аргументами. В качестве третьего аргумента будет выступать функциональный объект comparePersons().

```
sort( vectPtrsPers.begin(),
   vectPtrsPers.end(), comparePersons() );
```

Что касается нашего собственного функционального объекта comparePersons(), то его мы определяем в программе таким образом:

```
// функциональный объект для сравнивания содержимого //указателей на объекты person
class comparePersons
   {
   public:
   bool operator() (const person* ptrP1,
                    const person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
   };
```

При этом operator() имеет два аргумента, являющихся указателями на персональные данные, и сравнивает значения их содержимого, а не просто значения указателей.

Функциональный объект displayPerson()

Мы несколько необычным образом осуществляем вывод содержимого контейнера. Вместо простой итерации с поэлементным выводом мы используем функцию for_each() с собственным функциональным объектом в качестве третьего аргумента.

```
for_each( vectPtrsPers.begin(),
          vectPtrsPers.end(), displayPerson() );
```

Это выражение приводит к тому, что функциональный объект displayPerson() выводится для каждого элемента данных в векторе. Вот как выглядит объект displayPerson():

```
//функциональный объект для вывода персональных данных, //хранимых в указателях
class displayPerson
   {
   public:
   void operator() (const person* ptrP) const
      { ptrP->display(); }
   };
```

Таким образом, с помощью одного-единственного вызова функции на экран выводится содержимое всего вектора!

Функциональные объекты и поведение контейнеров

Функциональные объекты реально могут изменять поведение контейнеров. В программе SORTPTRS мы показали это. Например, с их помощью можно осуществлять сортировку данных в множестве указателей не по адресам последних, а по их содержимому, то есть сортировать реальные объекты, а не ссылки на них. Соответственно, при определении контейнера необходимо определить подходящий функциональный объект. И тогда не понадобятся никакие алгоритмы sort(). Этот подход мы обсудим ниже в одном из упражнений.

Резюме

В этой главе были очень кратко и бегло освещены вопросы, связанные с основами STL. Тем не менее, мы затронули большую часть тем, связанных с этим могучим инструментом C++. После внимательного прочтения этой главы можно смело приступать к практическому применению STL. Для более полного понимания всех возможностей Стандартной библиотеки шаблонов мы советуем читателю ознакомиться со специализированными изданиями, посвященными именно этой теме, благо их выпущено уже немало.

Теперь вы знаете, что STL состоит из трех основных компонентов: контейнеров, алгоритмов и итераторов. Контейнеры подразделяются на две группы: последовательные и ассоциативные. Последовательными являются векторы, списки и очереди с двусторонним доступом. Ассоциативные контейнеры — это, прежде всего, множества и отображения, а также тесно связанные с ними мультимножества и мультиотображения. Алгоритмы выполняют определенные операции над контейнерами, такие, как сортировка, копирование и поиск. Итераторы представляют собой разновидность указателей, применяющихся к элементам контейнеров, также они являются связующим звеном между алгоритмами и контейнерами.

Не все алгоритмы могут работать со всеми контейнерами. Итераторы используются для того, в частности, чтобы удостовериться, что данный алгоритм подходит данному контейнеру. Итераторы определяются только для некоторых видов контейнеров и могут являться аргументами алгоритмов. Если итератор контейнера не соответствует алгоритму, возникает ошибка компилятора.

Входные и выходные итераторы используются для подключения контейнеров непосредственно к потокам ввода/вывода, в результате этого возникает возможность работы контейнеров непосредственно с устройствами ввода/вывода! Специализированные итераторы дают возможность проходить контейнеры в обратном направлении, а также изменять поведение некоторых алгоритмов, например заставляя их вставлять данные, а не перезаписывать их.

Алгоритмы являются независимыми функциями, которые могут работать сразу со многими типами контейнеров. К тому же не следует забывать, что у контейнера есть свои собственные специфические методы. В некоторых случаях

Контейнеры и алгоритмы STL работают с данными любых типов, классов, вообще с чем угодно, лишь бы имелись в наличии перегружаемые операции типа <.

Поведение отдельных алгоритмов, таких, как find_if(), может быть изменено с помощью функциональных объектов. Последние реализуются с помощью классов, содержащих только один оператор ().

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Контейнер STL используется для:
   а) хранения объектов класса employee;
   б) хранения элементов таким образом, что доступ к ним осуществляется с крайне высокой скоростью;
   в) компилирования программ на C++;
   г) организации определенного способа хранения данных в памяти.
2. Перечислите последовательные контейнеры STL.
3. Двумя особо важными ассоциативными контейнерами STL являются ____________ и ______________.
4. Алгоритм STL — это:
   а) независимая функция для работы с контейнерами;
   б) связующий элемент между методами и контейнерами;
   в) функция, дружественная соответствующим классам контейнеров;
   г) метод соответствующих контейнеров.
5. Истинно ли утверждение о том, что одной из функций итераторов STL является связывание алгоритмов и контейнеров?
6. Алгоритм find():
   а) осуществляет поиск подходящих последовательностей элементов в двух контейнерах;
   б) осуществляет поиск контейнера, соответствующего указанному;
   в) в качестве первых двух аргументов использует итераторы;
   г) в качестве первых двух аргументов использует элементы контейнера.
7. Истинно ли утверждение о том, что алгоритмы могут использоваться только с контейнерами STL?
8. Диапазон часто задается в алгоритме двумя значениями типа ____________________.
9. Какая сущность зачастую используется для изменения поведения алгоритма?

10. Вектор является подходящим контейнером, если вы:

    а) собираетесь вставлять множество новых элементов в произвольные места контейнера;

    б) собираетесь вставлять новые элементы всегда в начало или конец контейнера;

    в) имеете индекс и хотите получить быстрый доступ к элементу с этим индексом;

    г) имеете ключевое значение элемента и хотите получить быстрый доступ к элементу по этому ключу.

11. Истинно ли утверждение о том, что метод back() удаляет элементы из конца контейнера?

12. Если вектор v определяется с помощью конструктора по умолчанию, а вектор w — с помощью конструктора с одним параметром и инициализируется размером 11 и если после этого в каждый из контейнеров вставляются три элемента с помощью push_back(), то метод size() возвратит значение ______ для контейнера v и ________ для контейнера w.

13. Алгоритм unique() удаляет все ______________ элементы из контейнера.

14. В очереди с двусторонним доступом:

    а) данные можно быстро вставлять (и удалять) в любую позицию;

    б) данные можно вставлять (удалять) в произвольную позицию, но это процесс относительно долгий;

    в) данные можно быстро вставлять (удалять) в концы очереди;

    г) данные можно вставлять (удалять) в концы очереди, но это процесс относительно долгий.

15. В итераторе _____________ конкретный элемент контейнера.

16. Истинно ли утверждение о том, что итератор всегда может сдвигаться как в прямом, так и в обратном направлении по контейнеру?

17. Необходимо использовать, по крайней мере, __________________ итератор для работы со списком.

18. Пусть iter — это итератор контейнера. Напишите выражение, имеющее значением объект, на который ссылается iter, и заставляющее затем сдвинуться iter на следующий элемент контейнера.

19. Алгоритм copy() возвращает итератор на:

    а) последний элемент, из которого производилось копирование;

    б) последний элемент, в который производилось копирование;

    в) элемент, располагающийся после последнего элемента, из которого производилось копирование;

    г) элемент, располагающийся после последнего элемента, в который производилось копирование.

20. Для того, чтобы использовать reverse_iterator, необходимо:
    а) начать с инициализации его значением end();
    б) начать с инициализации его значением rend();
    в) инкрементировать его для сдвига назад по контейнеру;
    г) декрементировать его для сдвига назад по контейнеру.
21. Истинно ли утверждение о том, что итератор back_inserter всегда приводит к тому, что новый элемент вставляется вслед за существующими?
22. Потоковые итераторы позволяют рассматривать файлы, а также такие устройства, как дисплей и клавиатура, в качестве ____________.
23. Что задает второй аргумент ostream_iterator?
24. В ассоциативном контейнере:
    а) значения хранятся в упорядоченном (отсортированном) виде;
    б) ключи упорядочены;
    в) сортировка всегда производится в алфавитном порядке или по возрастанию числовых значений;
    г) необходимо использовать алгоритм sort() для сортировки.
25. При определении множества необходимо указывать способ ____________.
26. Истинно ли утверждение о том, что во множестве метод insert() добавляет ключи с одновременной их сортировкой?
27. Отображение всегда хранит ____________ объектов (значений).
28. Истинно ли утверждение о том, что отображение может иметь два и более элементов с одинаковыми ключами?
29. Если в контейнере хранятся не объекты, а указатели на них, то:
    а) объекты не будут копироваться при их добавлении в контейнер;
    б) могут использоваться только ассоциативные контейнеры;
    в) невозможна сортировка по каким-либо атрибутам объекта;
    г) контейнеры будут занимать меньше памяти.
30. Если вы хотите, чтобы ассоциативный контейнер типа set производил автоматическую сортировку своих значений, можно определить порядок сортировки специальным функциональным объектом и указать его в числе ____________ контейнера.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

*1. Напишите программу, являющуюся примером применения алгоритма sort() к массиву типа float. Значения вводятся пользователем, результат работы выводится на экран.

*2. Примените алгоритм sort() к массиву слов, введенных пользователем, выведите результат. Используйте push_back() для добавления слов, а оператор [] и size() — для вывода их на экран.

*3. Начните с создания списка целых чисел. Используйте два обычных (не обратных) итератора: один для продвижения в прямом направлении, другой для продвижения в обратном направлении. Вставьте их в цикл while для переворачивания содержимого списка. Ради экономии нескольких выражений можно использовать алгоритм swap() (только при этом следует убедиться, что подход работает как с четными, так и с нечетными элементами). Чтобы посмотреть, как это делают настоящие профессионалы, взгляните на функцию reverse() в заголовочном файле ALGORITHM.

*4. Начните с класса person. Создайте мультимножество для хранения указателей на объекты этого класса. Определите его с помощью функционального объекта comparePersons, чтобы сортировка производилась автоматически по именам людей. Определите с полдюжины элементов, внесите их в мультимножество. Выведите его содержимое. Задайте имена людей таким образом, чтобы некоторые из них совпадали. Тем самым можно будет удостовериться в том, что мультимножество может хранить несколько объектов с одинаковым ключом.

5. Заполните массив четными числами, а множество — нечетными. С помощью алгоритма merge() объедините эти контейнеры в вектор. Выведите его содержимое, чтобы удостовериться, что слияние прошло успешно.

6. В упражнении 3 два обычных итератора использовались для переворачивания содержимого контейнера. Теперь используйте один прямой и один обратный контейнер для выполнения того же задания, на сей раз в применении к вектору.

7. В примере PLUSAIR мы показали применение версии accumulate() с четырьмя аргументами. Перепишите этот пример, используя версию с тремя аргументами. .

8. Алгоритм copy() можно использовать для копирования последовательностей в контейнере. Тем не менее нужно внимательно следить за тем, чтобы целевая последовательность не перекрывала исходную. Напишите программу, позволяющую производить копирование последовательностей внутри контейнера. Допустим, пользователь вводит значения first1, last1 и first2. Осуществляйте в программе проверку того, что последовательность, перекрывающая другую, сдвигается налево, а не направо. (Например, можно сдвигать некоторые данные из позиции 10 в позицию 9, но не в позицию 11.) Так делается потому, что copy() начинает работу с самого крайнего элемента слева.

9. В табл. 15.10 мы привели все предопределенные функциональные объекты C++. В программе PLUSAIR показали, как применять функциональный объект plus<>() с алгоритмом accumulate(). В том примере не было необходимости в передаче каких-либо аргументов в функциональный

объект, но иногда по логике работы программы это требуется. Оказывается, в данном случае нельзя просто указать аргумент в скобках, как мы привыкли делать. Вместо этого нужно использовать специальный «адаптер функции», называющийся bindldt или bind2nd, для связывания аргумента с функциональным объектом. Пусть, например, мы ищем строку SearchName в строковом контейнере names. В этом случае необходимо писать такое выражение:

```
ptr = find_if(names.begin(), names.end(), bind2nd(equal_to<string>(), SearchName) );
```

10. Здесь аргументами bind2nd, в свою очередь, являются equal_to<>() и SearchName. Это выражение возвращает итератор, указывающий на первую строку в контейнере, равную SearchName. Напишите программу, в которой используется подобное выражение для поиска строки в строковом контейнере. На экран должна выводиться позиция SearchName в контейнере.

11. Алгоритм copy_backward() используется для обхода проблемы, описанной в упражнении 7 (имеется в виду сдвиг последовательности влево, а не вправо при перекрытии последовательностей). Напишите программу, использующую и copy_backward(), и copy() для осуществления сдвига любой последовательности в любом направлении, независимо от перекрытия.

12. Напишите программу, копирующую один файл с целочисленными данными в другой. Используйте для этого потоковые итераторы. Пользователь должен ввести имена обоих файлов. Можно использовать цикл while. В этом цикле, наверное, будут читаться последовательно значения из входного итератора и сразу записываться в выходной, после чего будет производиться инкремент обоих итераторов. В качестве исходного можно взять файл ITER.DAT, созданный программой FOUTITER.

13. Частотная таблица представляет собой список слов и количество их вхождений в данном тексте. Напишите программу, создающую частотную таблицу для файла, имя которого введет пользователь. Для решения этой задачи хорошо подходит отображение, содержащее пары значений string-int. Можно использовать библиотечную функцию C под названием ispunct() (см. заголовочный файл CTYPE.H) для проверки правильности пунктуации, чтобы знать, где кончается каждое слово, и выделять его в подстроку методом substr(). Кроме того, функцией tolower() можно сделать все буквы слов строчными.

Глава 16
Разработка объектно-ориентированного ПО

- ✦ Эволюция процесса создания программного обеспечения
- ✦ Моделирование вариантов использования
- ✦ Предметная область программирования
- ✦ Программа LANDLORD: стадия развития
- ✦ От вариантов использования к классам
- ✦ Написание кода
- ✦ Взаимодействие с программой
- ✦ Заключение

Примеры программ, которые мы приводим в книге, настолько малы по объему, что не требуют какой-либо формализации процесса их разработки. Однако ситуация резко меняется, когда мы имеем дело с настоящей, масштабной программой, над созданием которой трудятся десятки или сотни программистов и в которой содержатся миллионы строчек исходного кода. В таких случаях очень важно четко следовать определенной концепции разработки ПО. В этой главе мы довольно сжато рассмотрим пример процесса создания программы, а затем покажем, как эту технологию применяют к настоящим программам.

В этой книге встречается множество примеров диаграмм UML. Тем не менее UML предназначен не для создания программы; это всего лишь язык визуального моделирования ее структуры. Но UML, как мы увидим далее, может играть ключевую роль в процессе работы над крупным проектом.

Эволюция процесса создания программного обеспечения

Идея формализации процесса разработки ПО развивалась в течение десятилетий. Мы лишь кратко рассмотрим основные вехи истории.

Процесс просиживания штанов

На заре человечества никаких формальностей не было. Программист обсуждал программу с потенциальными пользователями и сразу же бросался писать код. Это было, впрочем, вполне приемлемо для небольших программ.

Каскадный процесс

Когда программисты стали более серьезными людьми, они стали разбивать процесс разработки на несколько этапов, выполняемых последовательно. Эта идея была на самом деле позаимствована из производственных процессов. Этапы были таковы: анализ, планирование, кодирование и внедрение. Такая последовательность часто называлась *каскадной* или *водопадной моделью*, так как процесс всегда шел в одном направлении — от анализа к внедрению, как показано на рис. 16.1. Для реализации каждого из этапов стали привлекаться отдельные группы программистов, и каждая из них передавала результаты своего труда следующей.

Опыт показал, что водопадная модель имеет множество недостатков. Ведь подразумевалось, что каждый из этапов выполняется без ошибок или с минимальным их количеством. Так, конечно, почти не бывает в жизни. Каждая фаза вносила свои ошибки, их количество от этапа к этапу росло, как снежный ком, делая всю программу одной большой ошибкой.

К тому же в процессе разработки заказчик мог изменить свои требования, а по окончании этапа планирования уже сложно было вновь вернуться к нему. Оказывалось в итоге, что к моменту написания программа уже просто устаревала.

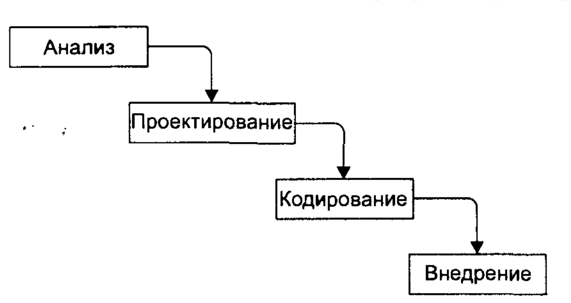


Рис. 16.1. Водопадная модель разработки ПО

Объектно-ориентированное программирование

Как мы уже говорили в главе 1, само по себе ООП создавалось для решения некоторых проблем, присущих процессу развития больших программ. Разумеется, процесс планирования при использовании ООП резко упрощается, так как объекты программы соответствуют объектам реального мира.

Но само по себе ООП не скажет нам, что должна делать программа, оно применимо уже после того, как определены цели и задачи проекта. Начальной, как и всегда, является фаза «инициализации», когда мы выясняем требования

заказчика и четко представляем себе нужды потенциальных пользователей. Только после этого можно начинать планирование объектно-ориентированной программы. Но как же нам сделать этот первый шаг?

Современные подходы

За последние годы появилось множество новых концепций разработки ПО. Они определяют последовательность действий и способы взаимодействия заказчиков, постановщиков задач, разработчиков и программистов. На сегодняшний день ни один язык моделирования не обладает той универсальностью, каковая присуща UML. Многие эксперты до сих пор не могут поверить, что один и тот же подход к разработке может быть применим к созданию проектов любых видов. Даже когда уже выбран какой-то процесс, может понадобиться, в зависимости от применения программы, довольно сильно его изменить. В качестве примера современного метода разработки ПО рассмотрим основные черты подхода, которому мы дадим название Унифицированный процесс.

Унифицированный процесс был создан теми же людьми, которые создали UML: Грэди Бучем (Grady Booch), Айвором Джекобсоном (Ivar Jacobson) и Джеймсом Рэмбо (James Rumbaugh). Иногда эту концепцию называют Рациональным унифицированным процессом (Rational Unified Process, по названию компании, в которой он был разработан) или Унифицированным процессом разработки ПО.

Унифицированный процесс разбивается на четыре этапа:

- начало;
- развитие;
- построение;
- передача.

На начальной стадии выявляются общие возможности будущей программы и ее осуществимость. Фаза оканчивается одобрением руководства проекта. На этапе развития планируется общая архитектура системы. Именно здесь определяются требования пользователей. На стадии построения осуществляется планирование отдельных деталей системы и пишется собственно код. В фазе внедрения система представляется конечным пользователям, тестируется и внедряется.

Все четыре фазы могут быть разбиты на ряд так называемых *итераций*. В частности, этап построения состоит из нескольких итераций. Каждая из них является подмножеством всей системы и отвечает определенным задачам, поставленным заказчиком. (Как мы увидим далее, итерации обычно соответствуют вариантам использования.) На рис. 16.2 показан Унифицированный процесс.

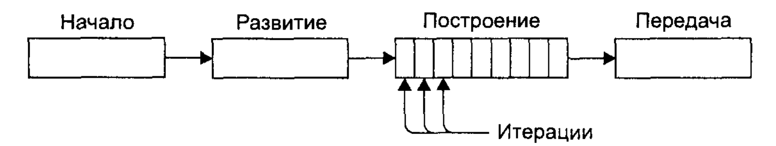


Рис. 16.2. Унифицированный процесс

Каждая итерация включает в себя свою собственную последовательность этапов анализа, планирования, реализации и тестирования. Итерации могут повторяться несколько раз. Целью итерации является создание работающей части системы.

В отличие от водопадного, Унифицированный процесс дает возможность вернуться на более ранние стадии разработки. Например, замечания, сделанные пользователями на стадии передачи, должны быть учтены, что приводит к пересмотру фазы построения и, возможно, фазы развития.

Следует отметить, что рассматриваемая концепция Унифицированного процесса может быть применена к любым типам программной архитектуры, а не только к проектам, в которых используются объектно-ориентированные языки. На самом деле, наверное, как раз слабой стороной этого подхода является не очень-то активное использование ООП.

Фаза развития обычно за основу берет технологию *вариантов использования* (use case). Это отправная точка детальной разработки системы. По этой причине Унифицированный процесс называют прецедентным. В следующем параграфе мы обсудим эту технологию, а затем применим ее к проекту, взятому нами в качестве примера.

Моделирование вариантов использования

Моделирование вариантов использования позволяет будущим пользователям самым активным образом участвовать в разработке ПО. При этом подходе за основу берется терминология пользователя, а не программиста. Это гарантирует взаимопонимание между заказчиками и системными инженерами.

В моделировании вариантов использования применяются две основные сущности: действующие субъекты и варианты использования. Давайте разберемся, кто есть кто.

Действующие субъекты

Действующий субъект — это в большинстве случаев просто тот человек, который будет реально работать с создаваемой нами системой. Например, действующим субъектом является покупатель, пользующийся торговым автоматом. Астроном, вводящий координаты звезды в программу автоматизации телескопа, — тоже действующий субъект. Продавец в книжном магазине, проверяющий по базе данных наличие книги, также может выступать в качестве действующего субъекта. Обычно этот человек инициирует некое событие в программе, какую-либо операцию, но может быть и «приемником» информации, выдаваемой программой. Кроме того, он может сопровождать и контролировать проведение операции.

На самом деле более подходящим названием, чем «действующий субъект» или «актер», является, возможно, «роль». Потому что один человек может в различных жизненных ситуациях играть разные роли. Частный предприниматель Пупкин может с утра быть продавцом в своем маленьком магазине, а вечером —

бухгалтером, вводящим данные о продажах за день. Наоборот, один действующий субъект может представляться разными людьми. В течение рабочего дня и Василий, и Василиса Пупкины являются продавцами в своем магазинчике.

Системы, находящиеся во взаимодействии с нашей, например другой компьютер локальной сети или web-сервер, могут быть действующими субъектами. Например, компьютерная система магазина книжной торговой сети может быть связана с удаленной системой в головном офисе. Последняя является действующим субъектом по отношению к первой.

При разработке большого проекта сложно бывает определить, какие именно действующие субъекты могут понадобиться. Разработчик должен рассматривать кандидатов на эти роли с точки зрения их взаимодействия с системой:

- вводят ли они данные;
- ожидают ли прихода информации от системы;
- помогают ли другим действующим субъектам.

Варианты использования

Вариант использования – это специальная задача, обычно инициируемая действующим субъектом. Она описывает единственную цель, которую необходимо в данный момент достичь. Примерами могут служить такие операции, как снятие денег с вклада клиентом банка, нацеливание телескопа астрономом, выяснение информации о наличии книги в книжном магазине его продавцом.

В большинстве ситуаций, как мы уже сказали, варианты использования генерируются действующими субъектами, но иногда их инициирует сама система. Например, компьютерная система ПетроЭлектроСбыта может прислать на ваш адрес напоминание о том, что пора заплатить за потребление электроэнергии. Электронная система (после встраивания ее в ваш «запорожец») может сообщить зажиганием контрольной лампы на панели приборов о перегреве двигателя.

В целом все, что должна делать система, должно быть описано с помощью вариантов использования на стадии ее разработки.

Сценарии

Вариант использования состоит в большинстве случаев из набора *сценариев*. В то время как вариант использования определяет цель операции, сценарий описывает способ достижения этой цели. Допустим, вариант использования состоит в том, что служащий книжного магазина запрашивает у системы местонахождение конкретной книги на складе. Существует несколько вариантов решения этой задачи (несколько сценариев):

- книга имеется в наличии на складе; компьютер выводит на экран номер полки, на которой она стоит;
- книга отсутствует, но система дает клиенту возможность заказать ее из издательства;
- книги не только нет на складе, ее нет вообще. Система информирует клиента о том, что ему не повезло.

Если со всей строгостью подходить к процессу разработки компьютерной системы, то каждый сценарий должен сопровождаться своей документацией, в которой описываются в деталях всевозможные события.

Диаграммы вариантов использования

С помощью UML можно строить диаграммы вариантов использования. Действующие субъекты представляются человечками, варианты использования — эллипсами. Прямоугольная рамка окружает все варианты использования, оставляя за своими пределами действующих субъектов. Этот прямоугольник называется *границей системы*. То, что находится внутри, — программное обеспечение, которое разработчик пытается создать. На рис. 16.3 показана диаграмма вариантов использования для компьютерной системы книжного магазина.

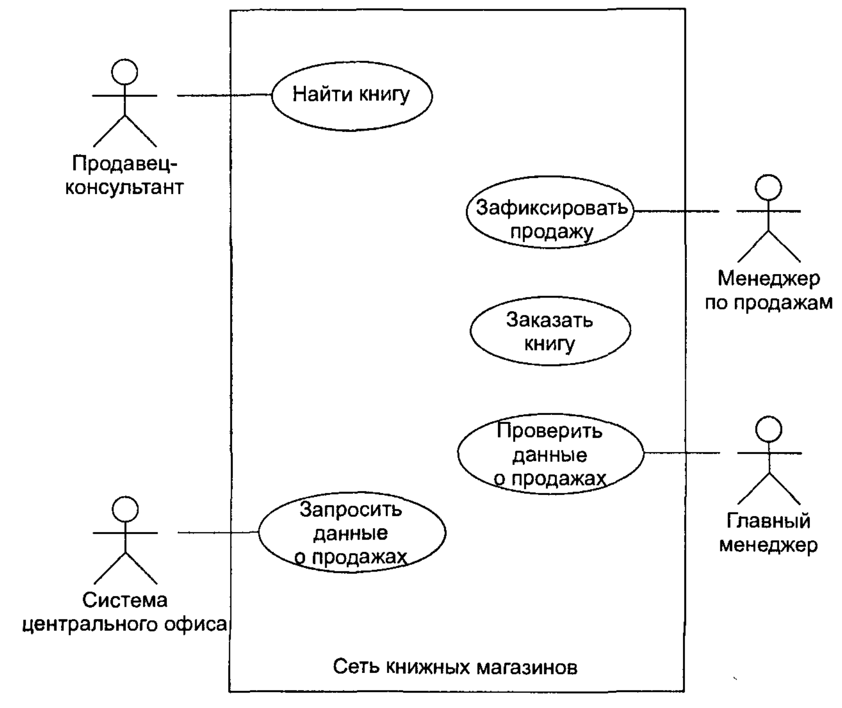


Рис. 16.3. Диаграмма вариантов использования для книжного магазина

На этой диаграмме линии, называемые *ассоциациями*, соединяют действующие субъекты с их вариантами использования. В общем случае ассоциации не являются направленными, и на линиях отсутствуют стрелочки, но можно их вставить для того, чтобы наглядно показать тот действующий субъект, который является инициатором варианта использования.

Мы предполагаем, что книжный магазин — это часть торговой сети, и бухгалтерия, и подобные функции выполняются в центральном офисе. Служащие магазина записывают покупку каждой книги и запрашивают информацию

о наличии товара и его местонахождении. Менеджер может просмотреть данные о том, какие книги проданы, и заказать в издательстве еще некоторое количество экземпляров. Действующими субъектами системы являются Продавец, Консультант, Менеджер и Система центрального офиса. Вариантами использования являются Регистрация продажи, Поиск книги, Заказ книги, Просмотр данных о реализации книг и Запрос данных о реализации книг.

Описания вариантов использования

На диаграмме вариантов использования нет места для размещения детального описания всех вариантов использования, поэтому приходится выносить описания за ее пределы. Для создания этих описаний можно использовать разные уровни формализации, в зависимости от масштабов проекта и принципов, которыми руководствуются разработчики. В большинстве случаев требуется детальное описание всех сценариев в варианте использования. Простейшей реализацией описания диаграммы вариантов использования является просто абзац-другой текста. Иногда используется таблица, состоящая из двух колонок: деятельность действующего субъекта и реакция на нее системы. Более формализованный вариант может включать в себя такие детали, как предусловие, постусловие, детальное описание последовательности шагов. Диаграмма UML, называемая *диаграммой действий*, являющаяся разновидностью блок-схемы, иногда используется как раз для того, чтобы графически изображать последовательность шагов в варианте использования.

Диаграммы вариантов использования и описания оных используются, прежде всего, при начальном планировании системы для обеспечения наилучшего взаимопонимания между заказчиками и разработчиками. Недаром используются такие наглядные значки в виде человечков, простые геометрические фигуры — ведь это этап разработки программы «на салфетке», то есть тот момент, когда пользователи и разработчики еще могут общаться друг с другом при помощи карандаша и огрызка бумаги. Но варианты использования, их диаграммы и описания бывают полезны и во время разработки программы. С ними можно сверяться для того, чтобы удостовериться, что программируются именно те действия, которые нужны, более того, они могут стать основой для тестирования и написания документации.

От вариантов использования к классам

Когда определены все действующие субъекты и варианты использования, процесс разработки плавно переходит из фазы развития в фазу построения программы. Это означает, что наметилась тенденция некого сдвига в сторону разработчиков от пользователей. С этого момента их тесное сотрудничество прекращается, поскольку они начинают говорить на разных языках. Первой проблемой, которую нам нужно решить, является создание и развитие классов, которые будут входить в программу.

Одним из подходов к именованию классов является использование имен существительных, встречающихся в кратких описаниях вариантов использования. Мы хотим, чтобы объекты классов программы соответствовали объектам реального мира, и эти существительные как раз являются именами тех сущностей, которые указал нам заказчик. Они являются кандидатами в классы, но, право слово, не из всех существительных могут получиться приемлемые классы. Нужно исключить слишком общие, тривиальные существительные, а также те, которые лучше представить в виде атрибутов (простых переменных).

После определения нескольких кандидатов в классы можно начинать думать о том, как они будут работать. Для этого стоит посмотреть на глаголы описаний вариантов использования. Частенько бывает так, что глагол становится тем сообщением, которое передается от одного объекта другому, или тем образом действия, который возникает между классами.

Диаграмму классов UML (она обсуждалась в предыдущих главах) можно использовать для того, чтобы показать классы и их взаимоотношения. Вариант использования реализуется последовательностью сообщений, посылаемых одними объектами другим. Можно использовать еще одну диаграмму UML, называемую *диаграммой взаимодействия*, для более детального представления этих последовательностей. На самом деле для каждого сценария варианта использования применяется своя диаграмма взаимодействия. В последующих параграфах мы рассмотрим примеры *диаграмм последовательностей*, являющихся разновидностью диаграмм взаимодействия.

Процесс разработки программ лучше описывать с использованием какого-нибудь живого примера, поэтому сейчас мы обратимся к рассмотрению процесса создания реальной программы. Но вы все-таки читаете книгу, а это не компакт-диск, поэтому программу пришлось сделать настолько маленькой, что, вообще говоря, спорным вопросом является целесообразность формализации процесса разработки. И все же даже такой пример должен помочь приоткрыть завесу тайны над понятиями, которые мы ввели выше.

Предметная область программирования

Программа, которую мы будем создавать в нашем примере, называется LANDLORD. Вы можете любить или не любить своего домовладельца или свою домомучительницу, но необходимо вполне представлять себе те данные, с которыми ему или ей приходится работать: плата за жилье и расходы. Вот такой незамысловатый бизнес. Его мы и будем описывать в нашей программе.

Представьте, что вы являетесь независимым экспертом по домомучительницам и C++, и к вам пришел заказчик по имени Степан Печкин. Печкин — мелкий собственник, в его ведении находится здание в Простоквашино, состоящее из 12 комнат, которые он сдает. Он хочет, чтобы вы написали программу, которая упростила бы его нелегкий труд по регистрации данных и распечатыванию отчетов о своей финансовой деятельности. Если вы сможете договориться со Степаном Маркелычем о цене, сроках и общем предназначении программы, можете считать, что вы включились в процесс разработки ПО.

Рукописные формы

На данный момент Степан Маркелович записывает всю информацию о своем доме вручную в старомодный гроссбух. Он показывает вам, какие формы используются для ведения дел:

- список жильцов;
- доходы от аренды;
- расходы;
- годовой отчет.

В *Списке жильцов* содержатся номера комнат и имена съемщиков, проживающих в них. Таким образом, это таблица из двух столбцов и 12 строк.

Доходы от аренды. Здесь хранятся записи о платежах съемщиков. В этой таблице содержится 12 столбцов (по числу месяцев) и по одной строке на каждую сдаваемую комнату. Всякий раз, получая деньги от жильцов, Печкин записывает заплаченную сумму в соответствующую ячейку таблицы, которая показана на рис. 16.4.

Месячный доход от аренды помещений

Номер комнаты	Янв	Фев	Март	Апр	Май	Июнь	Июль	Авг
101	696	695	695	695	695			
102	595	595	595	595	595			
103	810	810	825	825	825			
104	720	720	720	720	720			
201	680	680	680	680	680			
202	510	510	510	530	530			
203	790	790	790	790	790			
204	495	495	495	495	495			
301	585	585	585	585	585			
302	530	530	530	530	550			
303	810	810	810	810	810			
304	745	745	745	745	745			

Рис. 16.4. Доходы от аренды

Такая таблица наглядно показывает, какие суммы кем внесены.

В таблице *Расходы* записаны исходящие платежи. Она напоминает чековую книжку и содержит такие столбцы: дата, получатель (компания или человек, на чье имя выписывается чек) и сумма платежа. Кроме того, есть столбец, в который Печкин вносит виды или категории платежей: закладная, ремонт, коммунальные услуги, налоги, страховка и т. д. Таблица расходов показана на рис. 16.5.

В годовом отчете (рис. 16.6) используется информация как из таблицы доходов, так и из таблицы расходов для подсчета сумм, пришедших за год от клиентов и заплаченных в процессе ведения бизнеса. Суммируются все прибыли от всех жильцов за все месяцы. Также суммируются все расходы и записываются в соответствии с бюджетными категориями. Наконец, из доходов вычитаются расходы, в результате чего получается значение чистой годовой прибыли (или убытка).

Расходы

Дата	Получатель	Сумма	Категория
1/3	First Megabank	5187.30	Mortgage
1/8	City Water	963.10	Utilities
1/9	Steady State	4840.00	Insurance
1/15	P.G. & E.	727.23	Utilities
1/22	Sam's Hardware	54.81	Supplies
1/25	Ernie Glotz	150.00	Repairs
2/3	First Megabank	5187.30	Mortgage
2/7	City Water	845.93	Utilities
2/15	P.G. & E.	754.20	Utilities
2/18	Plotx & Skeems	1200.00	Legal Fees
3/2	First Megabank	5187.30	Mortgage
3/7	City Water	890.27	Utilities
3/10	Country of Springfield	9427.00	Property Texes
3/14	P.G. & E.	778.38	Utilities
3/20	Gotham Courier	26.40	Advertising
3/25	Ernie Glotz	450.00	Repairs
3/27	Acme Painting	600.00	Maintainance
4/3	First Megabank	5187.30	Mortgage

Рис. 16.5. Расходы

Годовой отчет

1		
2	INCOME	
3	Rent	102 264,00
4	TOTAL INCOME	102 264,00
5		
6	EXPENSES	
7	Mortgage	62 247,60
8	Property taxes	9 427,00
9	Insurance	4 840,00
10	Utilities	18 326,76
11	Supplies	1 129,23
12	Repairs	4 274,50
13	Maintenance	2 609,42
14	Legal fees	1 200,00
15	Landscaping	900,00
16	Advertising	79,64
17		
18	TOTAL EXPENSES	105 034,15
19		
20	NET PROFIT OR (LOSS)	(2 700,15)
21		

Рис. 16.6. Годовой отчет

Годовой отчет Степан Печкин составляет только в конце года, когда все доходы и расходы декабря уже известны. Наша компьютерная система будет выводить частичный годовой отчет в любое время дня и ночи за время, прошедшее с начала года.

Печкин уже человек немолодой и довольно консервативный, поэтому он, разумеется, просит сделать программу таким образом, чтобы внешний вид форм как можно более точно копировал таблички из его гроссбухов. Собственно говоря, основные задачи программы можно определить как ввод данных и вывод различных отчетов.

Допущения

Конечно, мы уже сделали несколько допущений и упрощений. Есть еще великое множество данных, связанных с ведением дел по сдаче в аренду помещений, таких, как залог за ущерб, амортизация, ипотечная выгода, доходы от запоздалых взносов (с начислением пени) и проката стиральных машин. Но мы не будем вдаваться в эти подробности.

Есть еще несколько отчетов, которые Печкин хочет видеть в программе. Например, отчет о чистой стоимости. Ну да, конечно, можно сделать такую программу, которая и перечисляла бы суммы по счетам, и могла бы работать как онлайновый магазин, но в этом случае со стороны заказчика было неразумно обратиться к услугам частного, независимого программиста. Кроме того, вообще-то есть и коммерческие программы для домовладельцев, и бухгалтерские системы, которые при желании можно приобрести. В общем, всяческие претензии Степана Маркелыча к неполноте нашей программы мы будем игнорировать.

Программа LANDLORD: стадия развития

Во время прохождения стадии развития должны происходить встречи потенциальных пользователей и реальных разработчиков для обсуждения того, что должна делать программа. В нашем примере С. М. Печкин является заказчиком и конечным пользователем системы, а мы с вами — тем экспертом, который будет и разрабатывать, и кодировать программу.

Действующие субъекты

Вначале нужно определить, кто будет являться действующими субъектами. Кто будет вводить информацию? Кто будет запрашивать? Будет ли кто-либо еще взаимодействовать с программой? Будет ли сама программа взаимодействовать с другими?

В примере LANDLORD с программой работает только один человек: домовладелец. Таким образом, один и тот же человек вводит информацию и просматривает ее в разных видах.

Даже в таком небольшом проекте можно представить себе еще каких-то действующих лиц. Это может быть счетовод, а сама наша программа может стать действующим субъектом, обращаясь к компьютерной системе налоговой службы. Для простоты мы не будем включать эти возможности в проект.

Варианты использования

Следующей группой, которую нужно описать, является группа действий, которые будет инициировать действующий субъект. В реальном проекте это может стать довольно объемной задачей, требующей длительного обсуждения и уточнения деталей. Здесь все не так сложно, и мы вполне можем практически сразу со-

ставить список вариантов использования, которые могут возникнуть при работе нашей программы. Все варианты использования отображены на диаграмме.

В нашем случае домовладельцу потребуется выполнять следующие действия:

- ✦ начать работу с программой;
- ✦ добавить нового жильца в список;
- ✦ ввести значение арендной платы в таблицу доходов от аренды;
- ✦ ввести значение в таблицу расходов;
- ✦ вывести список жильцов;
- ✦ вывести таблицу доходов от аренды;
- ✦ вывести таблицу расходов;
- ✦ вывести годовой отчет.

Диаграмма, которая в итоге получается, представлена на рис. 16.7.

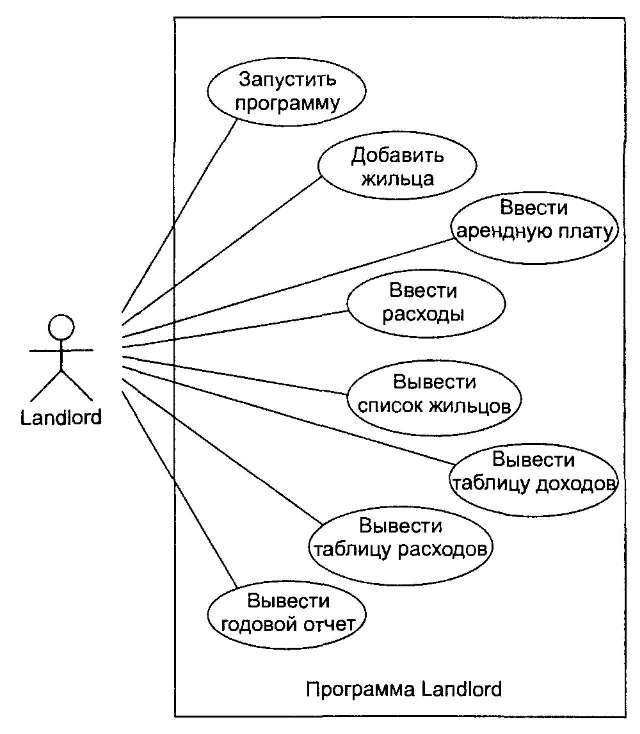


Рис. 16.7. Диаграмма вариантов использования для программы LANDLORD

Описание вариантов использования

Теперь неплохо было бы описать все варианты использования более детально. Как уже отмечалось, описания могут быть довольно формализованными и сложными. Но наш пример довольно прост, поэтому все, что нам нужно, — это небольшие описания в прозаической форме.

Начать программу

Это действие, казалось бы, слишком очевидно для того, чтобы о нем упоминать, но все же... Когда запускается программа, на экран должно выводиться меню, из которого пользователь может выбрать нужное действие. Это может называться экраном пользовательского интерфейса.

Добавить нового жильца

На экране должно отобразиться сообщение, в котором программа просит пользователя ввести имя жильца и номер комнаты. Эта информация должна заноситься в таблицу. Список автоматически сортируется по номерам комнат.

Ввести арендную плату

Экран ввода арендной платы содержит сообщение, из которого пользователь узнает, что ему необходимо ввести имя жильца, месяц оплаты, а также полученную сумму денег. Программа просматривает список жильцов и по номеру комнаты находит соответствующую запись в таблице доходов от аренды. Если жилец впервые вносит плату, в этой таблице создается новая строка и указанная сумма заносится в столбец того месяца, за который производится оплата. В противном случае значение вносится в существующую строку.

Ввести расход

Экран ввода расхода должен содержать приглашение пользователю на ввод имени получателя (или названия организации), суммы оплаты, дня и месяца, в который производится оплата, бюджетной категории. Затем программа создает новую строку, содержащую эту информацию, и вставляет ее в таблицу расходов.

Вывести список жильцов

Программа выводит на экран список жильцов, каждая строка списка состоит из двух полей: номер комнаты и имя жильца.

Вывести таблицу доходов от аренды

Каждая строка таблицы, которую выводит программа, состоит из номера комнаты и значения ежемесячной оплаты.

Вывести таблицу расходов

Каждая строка таблицы, которую выводит программа, состоит из значений месяца, дня, получателя, суммы и бюджетной категории платежа.

Вывести годовой отчет

Программа выводит годовой отчет, состоящий из:

- суммарной арендной платы за прошедший год;
- списка всех расходов по каждой бюджетной категории;
- результирующего годового баланса (доходы/убытки).

Сценарии

Мы уже упоминали о том, что вариант использования может состоять из нескольких сценариев. На данный момент мы описали только основной сценарий для каждого варианта использования. Это сценарий безошибочной работы, когда все идет гладко, и цель операции достигается точно таким образом, каким требуется. Однако необходимо предусмотреть более общие варианты развития событий в программе. В качестве примера можно привести случай попытки записи пользователем в таблицу жильцов второго жильца в занятую комнату.

Добавить нового жильца. Сценарий 2

На экране отображается экран ввода нового жильца. Введенный номер комнаты уже занят каким-то другим жильцом. Пользователю выводится сообщение об ошибке.

А вот еще один пример второго сценария для варианта использования. Здесь пользователь пытается ввести значение арендной платы для несуществующего жильца.

Ввести арендную плату. Сценарий 2

При вводе данных об арендной плате пользователь должен ввести имя жильца, месяц оплаты и ее сумму. Программа просматривает список жильцов, но не находит введенную фамилию. Выводится сообщение об ошибке.

В целях упрощения программы мы не будем развивать далее эти альтернативные сценарии, хотя в реальных проектах каждый дополнительный сценарий должен быть разработан с той же тщательностью, что и основной. Только так можно добиться того, чтобы программа действительно была применима в жизни.

Диаграммы действий UML

Диаграммы действий UML используются для моделирования вариантов использования. Этот тип диаграмм демонстрирует управляющие потоки от одних действий к другим. Он напоминает блок-схемы, которые существовали с самых первых дней развития технологий программирования. Но диаграммы действий очень хорошо формализованы и обладают дополнительными возможностями.

Действия показываются на диаграммах ромбовидными контурами. Линии, соединяющие действия, представляют собой переходы от одних действий к другим. Ветвление показано с помощью ромбиков с одним входом и двумя или более выходами. Как и на диаграмме состояний, можно поставить элементы, предназначенные для выбора одного из решений. Так же, как и там, можно задать начальное и конечное состояния, обозначаемые, соответственно, кружочком и кружочком в кольце.

На рис. 16.8 показан вариант использования Добавить нового жильца, включающий в себя оба сценария. Выбор ветви диаграммы зависит от того, занята или нет введенная пользователем комната. Если она уже занята, выводится сообщение об ошибке.

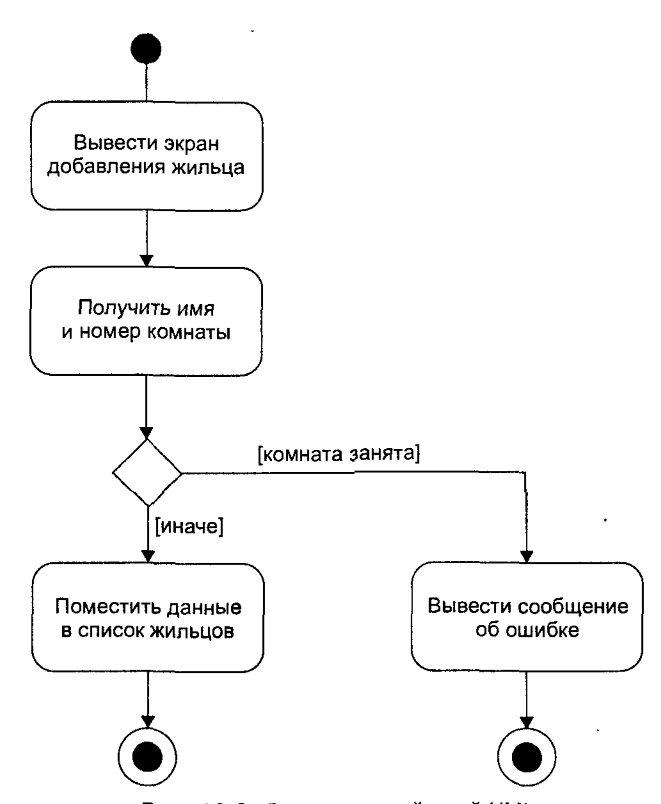


Рис. 16.8. Диаграмма действий UML

Диаграммы действий могут также использоваться для представления сложных алгоритмов, встречающихся в коде программы. В этом случае они практически идентичны блок-схемам. Они обладают некоторыми дополнительными возможностями, которые мы здесь не рассматриваем. Среди них, например, представление нескольких конкурирующих действий.

От вариантов использования к классам

Фаза построения начинается тогда, когда мы переходим к планированию структуры программы. Выберем классы. Для этого просмотрим список существительных из описаний вариантов использования.

Список существительных

Рассмотрим список всех существительных, взятых из описаний вариантов использования.

1. Экран интерфейса пользователя.
2. Жилец.

3. Экран ввода жильцов.
4. Имя жильца.
5. Номер комнаты.
6. Строка жильца.
7. Список жильцов.
8. Арендная плата.
9. Экран ввода арендной платы.
10. Месяц.
11. Сумма арендной платы.
12. Таблица доходов от аренды.
13. Строка арендной платы.
14. Расход.
15. Экран ввода расходов.
16. Получатель.
17. Размер платежа.
18. День.
19. Бюджетная категория.
20. Строка в таблице расходов.
21. Таблица расходов.
22. Годовой отчет.
23. Суммарная арендная плата.
24. Суммарные расходы по категориям.
25. Баланс.

Уточнение списка

По различным причинам многие существительные не смогут стать классами. Давайте произведем отбор только тех, которые могут претендовать на это высокое звание.

Мы выписали названия строк различных таблиц: строка жильцов, строка арендной платы, строка расходов. Иногда из строк могут получаться замечательные классы, если они составные или содержат сложные данные. Но каждая строка таблицы жильцов содержит данные только об одном жильце, каждая строка в таблице расходов — только об одном платеже. Классы жильцов и расходов уже существуют, поэтому мы осмелимся предположить, что нам не нужны два класса с одинаковыми данными, то есть мы не будем рассматривать строки жильцов и расходов в качестве претендентов на классы. Строка арендной платы, с другой стороны, содержит сведения о номере комнаты и массив из 12 платежей за аренду по месяцам. Она отсутствует в таблице до тех пор, пока не будет сделан первый взнос в текущем году. Последующие платежи вносятся

в уже существующие строки. Это ситуация более сложная, чем с жильцами и расходами, поэтому, так и быть, сделаем строку арендной платы классом. Тогда класс арендной платы как таковой не будет содержать ничего, кроме суммы платежа, поэтому это существительное превращать в класс мы не станем.

Программа может порождать значения в годовом отчете из таблицы арендной платы и таблицы расходов, поэтому, наверное, не стоит сумму арендных платежей, а также суммарные расходы по категориям и баланс делать отдельными классами. Они являются просто результатами вычислений.

Итак, составим список классов, которые мы с вами придумали.

1. Экран пользовательского интерфейса.
2. Жилец.
3. Экран ввода жильцов.
4. Список жильцов.
5. Экран ввода арендной платы.
6. Таблица доходов от аренды.
7. Строка таблицы доходов от аренды.
8. Расход.
9. Экран ввода расходов.
10. Таблица расходов.
11. Годовой отчет.

Определение атрибутов

Многие существительные, которым отказано в регистрации в кандидаты классов, будут кандидатами в атрибуты (компонентные данные) классов. Например, у класса Жильцы будут такие атрибуты: Имя жильца, Номер комнаты. У класса Расходы: Получатель, Месяц, День, Сумма, Бюджетная категория. Большинство атрибутов может быть определено таким путем.

От глаголов к сообщениям

Теперь посмотрим, что нам дают варианты использования для выяснения того, какими сообщениями будут обмениваться классы. Поскольку сообщение — это, по сути дела, вызов метода в объекте, то определение сообщений сводится к определению методов класса, принимающего то или иное сообщение. Как и в случае с существительными, далеко не каждый глагол становится кандидатом в сообщения. Некоторые из них, вместо приема данных от пользователей, связываются с такими операциями, как вывод информации на экран, или с какими-либо еще действиями.

В качестве примера рассмотрим описание варианта использования Вывести список жильцов. Курсивом выделены глаголы.

> Программа *выводит* на экран список жильцов, каждая строка списка *состоит* из двух полей: номер комнаты и имя жильца.

Под словом «программа» мы на самом деле имеем в виду экран пользовательского интерфейса, следовательно, слово «выводит» означает, что объект «экран пользовательского интерфейса» посылает сообщение объекту Список жильцов (то есть вызывает его метод). В сообщении содержится указание вывести самого себя на экран. Несложно догадаться, что метод может называться, например, display().

Глагол «состоит» не относится ни к какому сообщению. Он просто примерно определяет состав строки объекта Список жильцов.

Рассмотрим более сложный пример: вариант использования Добавить нового жильца:

> На экране должно отобразиться сообщение, в котором программа *просит* пользователя *ввести* имя жильца и номер комнаты. Эта информация должна *заноситься* в таблицу. Лист автоматически *сортируется* по номерам комнат.

Глагол «отобразиться» в данном случае будет обозначать следующее. Экран пользовательского интерфейса должен послать сообщение классу «экран ввода жильцов», приказывая ему вывести себя и получить данные от пользователя. Это сообщение может быть вызовом метода класса с именем, например getTenant().

Глаголы «просит» и «ввести» относятся к взаимодействию класса «экран ввода жильцов» с пользователем. Они не становятся сообщениями в объектном смысле. Просто-напросто getTenant() выводит приглашение и записывает ответы пользователя (имя жильца и номер комнаты).

Глагол «заноситься» означает, что объект класса Экран ввода жильцов посылает сообщение объекту класса Список жильцов. Возможно, в качестве аргумента используется новый объект класса Жильцы. Объект Список жильцов может затем вставить этот новый объект в свой список. Эта функция может иметь имя типа insertTenant().

Глагол «сортируется» — это никакое не сообщение и вообще не вид взаимодействия объектов. Это просто описание списка жильцов.

На рис. 16.9 показан вариант использования Добавить нового жильца и его связи с описанными выше сообщениями.

Когда мы начнем писать код, то обнаружим, что некоторые действия остались вне нашего рассмотрения в этом варианте использования, но требуются программе. Например, нигде не говорится о создании объекта Жилец. Тем не менее, наверное, понятно, что Список Жильцов содержит объекты типа Жилец, а последние должны быть созданы до их внесения в список. Итак, системный инженер решает, что метод getTenant() класса «экран ввода жильцов» — это подходящее место для создания объекта Жилец, вставляемого в список жильцов.

Все прочие варианты использования должны быть проанализированы аналогичным образом, чтобы можно было создать основу для связывания классов. Обратите внимание, что мы все еще используем имена классов, совпадающие со словами или словосочетаниями вариантов использования. Когда мы начнем писать код, нам, конечно, придется переименовать их во что-либо более приемлемое (имена должны состоять из одного слова).

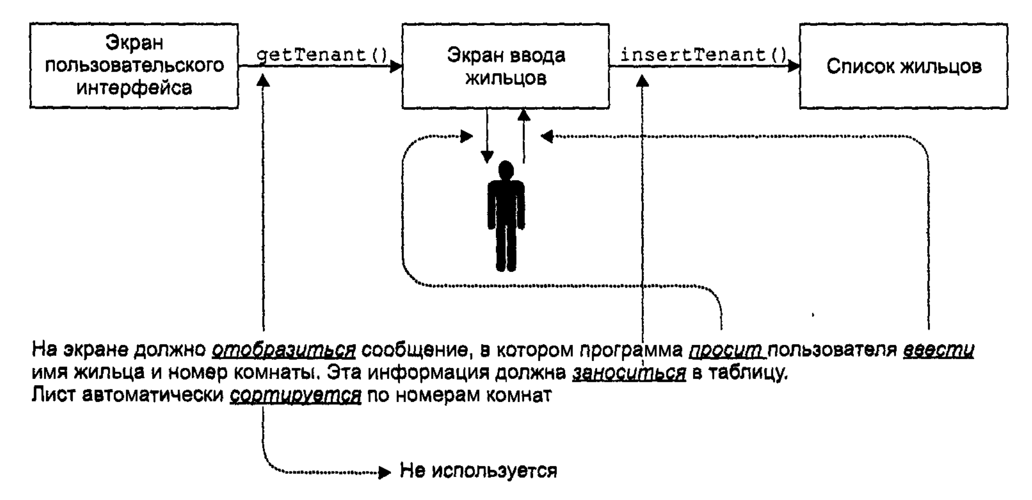


Рис. 16.9. Глаголы варианта использования Добавить Нового жильца

Диаграмма классов

Зная, какие классы будут включены в нашу программу и как они будут между собой связаны, мы сможем построить диаграмму классов. В предыдущих главах мы уже видели примеры таких диаграмм.

На рис. 16.10 показана диаграмма классов программы LANDLORD.

Диаграммы последовательностей

Прежде чем начать кодировать, было бы логично разобраться более детально, как выполняется каждый шаг каждого варианта использования. Для этого можно разработать диаграмму последовательностей UML. Это один из двух типов диаграмм взаимодействия UML (второй тип — совместная диаграмма). И на той, и на другой отображается, каким обэразом события разворачиваются во времени. Просто диаграмма последовательностей более наглядно изображает процесс течения времени. На ней вертикальная ось — это время. Оно «начинается» вверху и течет сверху вниз по диаграмме. Наверху находятся имена объектов, которые будут принимать участие в данном варианте использования. Действие обычно начинается с того, что объект, расположенный слева, посылает сообщение объекту, расположенному справа. Обычно чем правее расположен объект, тем ниже его значимость для программы или больше его зависимость от других.

Обратите внимание на то, что на диаграмме показаны не классы, а объекты. Говоря о последовательностях сообщений, необходимо упомянуть, что сообщения пересылаются именно между объектами, не между классами. На диаграммах UML названия объектов отличаются от названий классов наличием подчеркивания.

Линией жизни называется пунктирная линия, уходящая вниз от каждого объекта. Она показывает, когда объект начинает и заканчивает свое существование. В том месте, где объект удаляется из программы, линия жизни заканчивается.

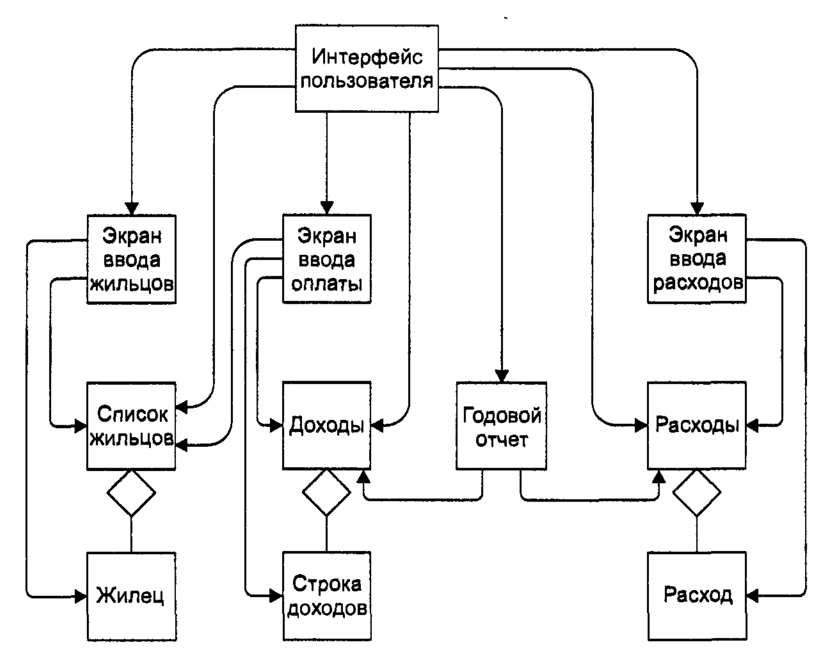


Рис. 16.10. Диаграмма классов для программы LANDLORD

Диаграмма последовательностей для варианта использования Начать программу

Обратим внимание на некоторые из диаграмм последовательностей нашей программы. Начнем с самой простой из них. На рис. 16.11 показана диаграмма для варианта использования Начать программу.

При запуске программы определяется userInterface — класс поддержки экрана пользовательского интерфейса, который мы так долго обсуждали, говоря о вариантах использования. Допустим, программа создает единственный объект класса под названием theUserInterface. Именно этот объект порождает все варианты использования. Он появляется слева на диаграмме последовательностей. (Как видите, мы на этом этапе уже перешли к нормальным именам классов, принятым при написании кода.)

Когда вступает в работу объект theUserInterface, первое, что он делает, он создает три основные структуры данных в программе. Это объекты классов TenantList, rentRecord и ExpenceRecord. Получается, что они рождаются безымянными, так как для их создания используется new. Имена имеют только указатели на них. Как же нам их назвать? К счастью, как мы убедились на примере объектных диаграмм, UML предоставляет несколько способов именования объектов. Если настоящее имя неизвестно, можно использовать вместо него двоеточие с именем класса (:tenantList). На диаграмме подчеркивание имени и двоеточие перед ним напоминает о том, что мы имеем дело с объектом, а не с классом.

Вертикальная позиция прямоугольника с именем объекта указывает на тот момент времени, когда объект был создан (первым создается объект класса

tenantList). Все объекты, которые вы видите на диаграмме, продолжают существовать все время, пока программа стоит на исполнении. Шкала времени, строго говоря, рисуется не в масштабе — она предназначена только лишь для того, чтобы показать взаимосвязь различных событий.

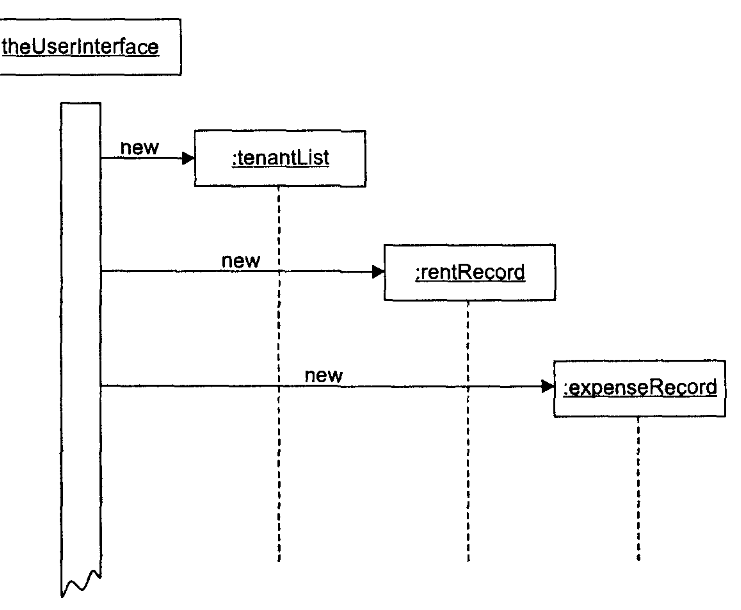


Рис. 16.11. Диаграмма последовательностей для варианта использования Начать программу

Горизонтальные линии представляют собой сообщения (то есть вызовы методов). Сплошная стрелочка говорит о нормальном синхронном вызове функции, открытая — об асинхронном событии.

Прямоугольник, расположенный под theUserInterface, называется *блоком активности* (или *центром управления*). Он показывает, что расположенный над ним объект является активным. В обычной процедурной программе, такой, как наша (LANDLORD), «активный» означает, что метод данного объекта либо исполняется сам, либо вызвал на исполнение другую функцию, которая еще не завершила свою работу. Три других объекта на этой диаграмме не являются активными, потому что theUserInterface еще не послал им активизирующих сообщений.

Диаграмма последовательностей для варианта использования Вывод списка жильцов

Еще один пример диаграммы последовательностей представлен на рис. 16.12. На рисунке изображена работа варианта использования Вывод списка жильцов. Возвращения значений функциями показаны прерывистыми линиями. Обратите внимание: объекты активны только тогда, когда вызван какой-либо их метод. Сверху над линией сообщения может быть указано имя вызываемого метода.

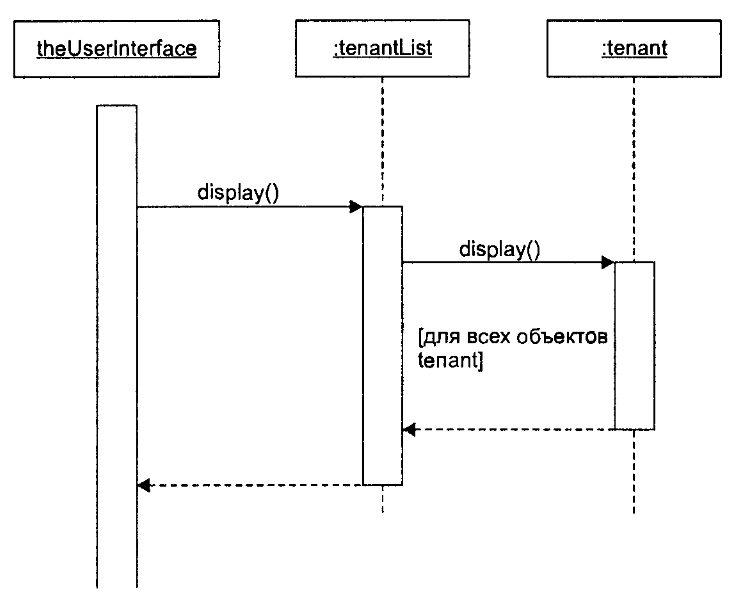


Рис. 16.12. Диаграмма последовательностей для варианта использования Вывод списка жильцов

Что мы видим? Объект theUserInterface дает задание объекту tenantList вывести себя на экран (вызовом его метода display()), а тот, в свою очередь, выводит все объекты класса tenant. Звездочка означает, что сообщение будет посылаться циклически, а фраза в квадратных скобках [для всех объектов tenant] сообщает условие повторения. (В программе, на самом деле, мы будем использовать cout << вместо функции display().)

Диаграмма последовательностей для варианта использования Добавить нового жильца

Наконец, последним примером диаграммы последовательностей, который мы здесь приводим, является диаграмма для варианта использования Добавить нового жильца. Она показана на рис. 16.13. Сюда мы включили самого домовладельца в виде объекта, который определяет различные действия. У него есть свой собственный блок активности. С помощью этого объекта можно очень хорошо показать процесс взаимодействия пользователя с программой.

Пользователь сообщает программе, что он желает добавить нового жильца. Объект theUserInterface создает новый объект класса tenantInputScreen. В этом объекте есть методы, позволяющие получить от пользователя данные о жильце, создать новый объект типа tenant и вызвать метод объекта класса tenantList для добавления в список вновь созданного жильца. Когда все это проделано, объект theUserInterface удаляется. Большая буква «X» в конце линии жизни tenantInputScreen говорит о том, что объект удален.

Диаграммы последовательностей, примеры которых мы здесь привели, имеют дело только с главными сценариями каждого варианта использования. Сущест-

вуют, конечно же, способы показать на диаграммах и несколько сценариев, но можно и для каждого сценария создать свою диаграмму.

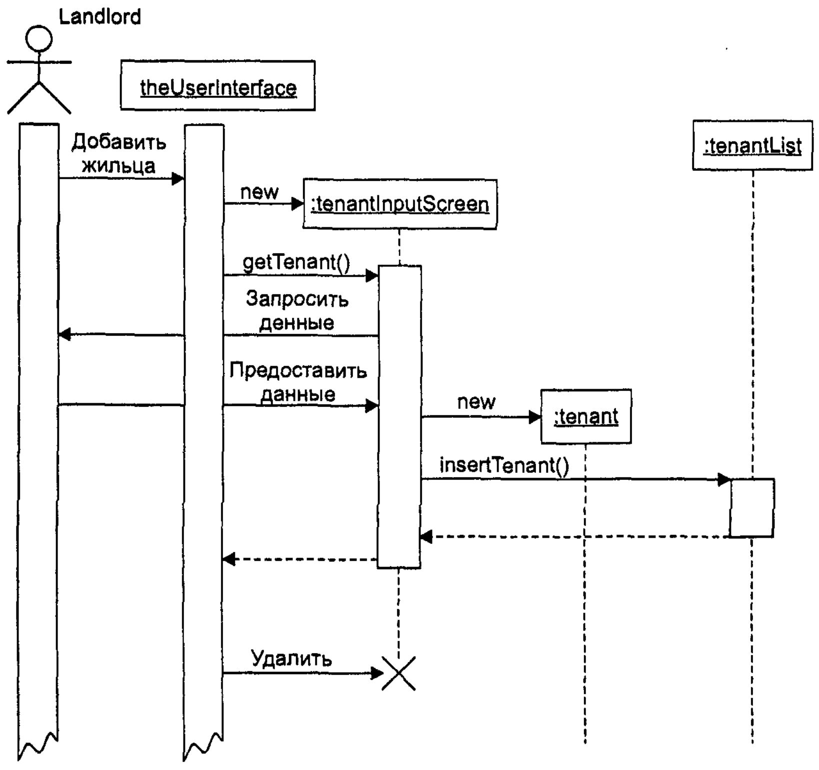


Рис. 16.13. Диаграмма последовательностей для варианта использования Добавить нового жильца

Недостаток места в книге не позволяет нам привести все примеры диаграмм последовательностей, но, кажется, и так уже показано достаточно для их понимания.

Написание кода

Наконец, вооружившись диаграммами вариантов использования, детальными описаниями вариантов использования, диаграммой классов, диаграммами последовательностей и предварительными планами создания программы, можно запустить компилятор и начать писать собственно код. Это вторая часть фазы построения.

Варианты использования, определенные на этапе развития, становятся итерациями на новом этапе (см. рис. 16.2). В большом проекте каждая итерация может производиться разными группами программистов. Все итерации должны проектироваться по отдельности, а их результаты — предоставляться заказчику

для внесения добавлений и исправлений. В нашей небольшой программе, впрочем, эти лишние сложности ни к чему.

Заголовочный файл

Ура, мы добрались до написания таких родных и привычных файлов с кодами. Лучше всего начинать с заголовочного (.H) файла, в котором следует определить только интерфейсные части классов, но не подробности реализации. Как говорилось ранее, объявления в заголовочном файле — это общедоступная часть классов. Тела функций расположены в .CPP-файлах, называются реализацией и пользователям недоступны.

Написание заголовочного файла — это промежуточный шаг между планированием и обыкновенным кодированием методов. Рассмотрим содержимое заголовочного файла LANDLORD.H.

Листинг 16.1. Заголовочный файл к программе LANDLORD[1]

```
//landlord.h
//заголовочный файл landlord.cpp - содержит объявления
//классов и т.п.
#pragma warning (disable:4786) //для множеств (только
                               // компиляторы microsoft)
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <algorithm>           //для sort()
#include <numeric>             //для accumulate()
using namespace std;
///////////////////глобальные методы/////////////////////
void getaLine(string& inStr);  // получение строки текста
char getaChar();               // получение символа

///////////////////класс tenant (жильцы)/////////////////
class tenant
  {
  private:
  string name;        // имя жильца
  int aptNumber;      // номер комнаты жильца
  // здесь может быть прочая информация о жильце
  // (телефон и т.п.)

  public:
  tenant(string n, int aNo);
  ~tenant();
  int getAptNumber();
  // нужно для использования во множестве
  friend bool operator < (const tenant&, const tenant&);
  friend bool operator == (const tenant&, const tenant&);
  // для операций ввода-вывода
  friend ostream& operator << (ostream&, const tenant&);
};  // конец объявления класса tenant
```

продолжение

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http ://www.piter.com/download.

Листинг 16.1 (*продолжение*)

```
////////////////////класс compareTenants/////////////////
class compareTenants          //функциональный обьект для
                              // сравнения имен жильцов
  {
  public:
  bool operator () (tenant*. tenant*) const:
  }:

//////////////////класс tenantList///////////////////////
class tenantList
  {
  private:
  // установить указатели на жильцов
  set<tenant*. compareTenants> setPtrsTens:
  set<tenant*. compareTenants>::iterator iter:

  public:
  ~tenantList():              // деструктор
                              // (удаление жильцов)
  void insertTenant(tenant*): // внесение жильца в список
  int getAptNo(string):       // возвращает номер комнаты
  void display():             // вывод списка жильцов
   }:
// конец обьявления класса tenantList

///////////////класс tenantInputScreen///////////////////
class tenantInputScreen
  {
  private:
  tenantList* ptrTenantList:
  string tName:
  int aptNo:

  public:
  tenantInputScreen(tenantList* ptrTL) : ptrTenantList(ptrTL)
     { /* тут пусто */ }
  void getTenant():
  }:  //конец класса tenantInputScreen

///////////////////класс rentRow/////////////////////////
// одна строка таблицы прибыли: адрес и 12 месячных оплат
class rentRow
  {
  private:
  int aptNo:
  float rent[12]:

  public:
  rentRow(int):               // конструктор с одним
                              // параметром
  void setRent(int. float):   // запись платы за месяц
  float getSumOfRow():        // сумма платежей
                              // из одной строки
  // нужно для сохранения во множестве
  friend bool operator < (const rentRow&. const rentRow&):
  friend bool operator == (const rentRow&. const rentRow&):
```

```
  // для вывода
  friend ostream& operator << (ostream&, const rentRow&);
  }; // конец класса rentRow

////////////////////////////////////////////////////////////
class compareRows               //функциональный объект сравнения
                                //объектов rentRows
  {
  public:
    bool operator () (rentRow*, rentRow*) const;
  };

////////////////////класс rentRecord//////////////////////
class rentRecord
  {
  private:
  // множество указателей на объекты rentRow (по одному на
  //  жильца)
  set<rentRow*, compareRows> setPtrsRR;
  set<rentRow*, compareRows>::iterator iter;

  public:
  ~rentRecord();
  void insertRent(int, int, float);
  void display();
  float getSumOfRents();        // сумма всех платежей
   }; // конец класса rentRecord

////////////////////класс rentInputScreen/////////////////
class rentInputScreen
  {
  private:
  tenantList* ptrTenantList;
  rentRecord* ptrRentRecord;

  string renterName;
  float rentPaid;
  int month;
  int aptNo;

  public:
  rentInputScreen(tenantList* ptrTL, rentRecord* ptrRR) :
                    ptrTenantList(ptrTL), ptrRentRecord(ptrRR)
     { /*тут пусто*/ }
  void getRent();               //арендная плата одного
                                // жильца за один месяц
  }; // конец класса rentInputScreen

//////////////////////класс expense/////////////////////
class expense
  {
  public:
  int month, day;
  string category, payee;
  float amount;
  expense()
    { }
```

продолжение

Листинг 16.1 (*продолжение*)

```
  expense(int m, int d, string c, string p, float a) :
       month(m), day(d), category(c), payee(p), amount(a)
    { /* тут пусто! */ }
  // нужно для хранения во множестве
  friend bool operator < (const expense&, const expense&);
  friend bool operator == (const expense&, const expense&);
  // нужно для вывода
  friend ostream& operator << (ostream&, const expense&);
  };  // конец класса expense
////////////////////////////////////////////////////////////////
class compareDates //функциональный объект сравнения затрат
  {
  public:
    bool operator () (expense*, expense*) const;
  };
////////////////////////////////////////////////////////////////
class compareCategories        //функциональный объект
                               // сравнения затрат
  {
  public:
    bool operator () (expense*, expense*) const;
  };

///////////////////класс expenseRecord///////////////////
class expenseRecord
  {
  private:
  //вектор указателей на расходы
  vector<expense*> vectPtrsExpenses;
  vector<expense*>::iterator iter;

  public:
  ~expenseRecord();
  void insertExp(expense*);
  void display();
  float displaySummary();      // нужно для годового
                               // отчета
  };  // конец класса expenseRecord

////////////////класс expenseInputScreen//////////////////
class expenseInputScreen
  {
  private:
  expenseRecord* ptrExpenseRecord;

  public:
  expenseInputScreen(expenseRecord*);
  void getExpense();
  };  // конец класса expenseInputScreen

/////////////////класс annualReport/////////////////////
class annualReport
  {
  private:
```

```
  rentRecord* ptrRR;
  expenseRecord* ptrER;
  float expenses, rents;

  public:
  annualReport(rentRecord*, expenseRecord*);
  void display();
  }; // конец класса annualReport

//////////////////конец userInterface//////////////////////
class userInterface
  {
   private:
  tenantList*          ptrTenantList;
  tenantInputScreen*   ptrTenantInputScreen;
  rentRecord*          ptrRentRecord;
  rentInputScreen*     ptrRentInputScreen;
  expenseRecord*       ptrExpenseRecord;
  expenseInputScreen*  ptrExpenseInputScreen;
  annualReport*        ptrAnnualReport;
  char ch;

  public:
  userInterface();
  ~userInterface();
  void interact();
  }; // конец класса userInterfac
//////////////////конец файла landlord.h///////////////////
```

Объявления классов

Объявлять классы — это просто. Большинство объявлений вырастают напрямую из классов, созданных с помощью взятых из описаний вариантов использования существительных, и отображаются на диаграмме классов. Только лишь имена нужно сделать однословными из многословных. Например, имя Список жильцов (Tenant List) превращается в TenantList.

В заголовочном файле было добавлено еще несколько вспомогательных классов. Впоследствии окажется, что мы храним указатели на объекты в разных типах контейнеров STL. Это означает, что мы должны сравнивать объекты этих контейнеров, как описано в главе 15 «Стандартная библиотека шаблонов (STL)». Объектами сравнения на самом деле являются классы compareTenants, compareRows, compareDates и compareCategories.

Описания атрибутов

Как уже было замечено выше, многие атрибуты (методы) для каждого из классов вырастают из тех существительных, которые сами не стали классами. Например, name и aptNumber стали атрибутами класса tenant.

Прочие атрибуты могут быть выведены из ассоциаций в диаграмме классов. Ассоциации могут определять те атрибуты, которые являются указателями или ссылками на другие классы. Это объясняется невозможностью ассоциировать что-то с чем-то, что находится неизвестно где. Таким образом, у класса rentInputScreen появляются атрибуты ptrTenantList и ptrRentRecord.

Составные значения (агрегаты)

Агрегатные связи показаны в трех местах на диаграмме классов. Обычно агрегаты выявляют те контейнеры, которые являются атрибутами агрегирующего класса (то есть «целого» класса, содержащего «части»).

Ни по описаниям вариантов использования, ни по диаграмме классов невозможно угадать, какого рода контейнеры должны использоваться для этих агрегатов. Вам как программистам придется самим всякий раз выбирать подходящий контейнер для каждого составного значения — будь то простой массив, контейнер STL или что-либо еще. В программе LANDLORD мы сделали такой выбор:

- класс tenantlist содержит STL-множество указателей на объекты класса tenant;
- класс rentRecord содержит множество указателей на объекты класса rentRow;
- класс expenseRecord содержит вектор указателей на объекты класса expense.

Для tenantlist и rentRecord мы выбрали множества, так как основным параметром является быстрый доступ к данным. Для expenseRecord выбран вектор, потому что нам важно осуществлять быструю сортировку и по дате, и по категории, а векторы позволяют сортировать данные наиболее эффективно.

Во всех агрегатах мы предпочли хранить указатели вместо самих объектов, чтобы избежать излишнего копирования данных в памяти. Хранение самих объектов следует применять в тех случаях, когда объектов мало и они небольшие. Конечно, большой разницы в скорости на примере каких-то 12 экземпляров объекта мы не увидим, но в принципе об эффективности метода хранения данных следует задумываться всегда.

Исходные .cpp файлы

В исходных файлах содержатся тела методов, которые были объявлены в заголовочном файле. Написание кода этих методов должно начинаться только на этом этапе разработки и ни шагом раньше, потому что только сейчас мы знаем имя каждой функции, ее предназначение и даже, возможно, можем предугадать аргументы, передаваемые ей.

Мы отделили зерна от плевел: main() храним в одном коротеньком файле LORDAPP.CPP, а определения функций, объявленных в заголовочном файле, — в другом. В секции main() создается объект userInterface и вызывается метод interact(). Приведем файл, в котором хранится main().

Листинг 16.2. Программа LORDAPP.CPP

```
// lordApp.cpp
// файл, поставляемый клиенту.
#include "landlord.h"

int main()
  {
  userInterface theUserInterface;

  theUserInterface.interact();
```

```
  return 0;
  }
///////////////////конец файла lordApp.cpp////////////////
```

Ну и наконец, рассмотрим файл LANDLORD.CPP, в котором содержатся все определения методов.

Листинг 16.3. Программа LANDLORD.CPP

```
void getaLine(string& inStr)      // получение строки текста
  {
  char temp[21];
  cin.get(temp, 20, '\n');
  cin.ignore(20, '\n');
  inStr = temp;
  }
//---------------------------------------------------------
char getaChar()                   // получение символа
  {
  char ch = cin.get();
  cin.ignore(80, '\n');
  return ch;
  }
//---------------------------------------------------------

////////////////методы класса tenant//////////////////////
tenant::tenant(string n, int aNo) : name(n), aptNumber(aNo)
  { /* тут пусто */ }
//---------------------------------------------------------
tenant::~tenant()
  { /* тут тоже пусто */ }
//---------------------------------------------------------
int tenant::getAptNumber()
  { return aptNumber; }
//---------------------------------------------------------
bool operator < (const tenant& t1, const tenant& t2)
  { return t1.name < t2.name; }
//---------------------------------------------------------
bool operator == (const tenant& t1, const tenant& t2)
  { return t1.name == t2.name; }
//---------------------------------------------------------
ostream& operator << (ostream& s, const tenant& t)
  { s << t.aptNumber << '\t' << t.name << endl;  return s; }
//---------------------------------------------------------

///////////метод класса tenantInputScreen/////////////////
void tenantInputScreen::getTenant()   // получение данных о
                                      // жильцах
  {
  cout << "Введите имя жильца (Дядя Федор): ";
  getaLine(tName);
  cout << "Введите номер комнаты (101): ";
  cin >> aptNo;
  cin.ignore(80, '\n');               // создать жильца
  tenant* ptrTenant = new tenant(tName, aptNo);
  ptrTenantList->insertTenant(ptrTenant); // занести в
                                          // список жильцов
  }
```

продолжение

Листинг 16.3 (*продолжение*)

```
///////////////////////////////////////////////////////////
bool compareTenants::operator () (tenant* ptrT1.
                                  tenant* ptrT2) const
  { return *ptrT1 < *ptrT2; }
//---------------------------------------------------------

////////////////методы класса tenantList///////////////////
tenantList::~tenantList()      // деструктор
  {
  while( !setPtrsTens.empty() ) // удаление всех жильцов,
      {                         // удаление указателей из
                                // множества
    iter = setPtrsTens.begin();
    delete *iter;
    setPtrsTens.erase(iter);
    }
  }  // end ~tenantList()
//---------------------------------------------------------
void tenantList::insertTenant(tenant* ptrT)
  {
  setPtrsTens.insert(ptrT);     //вставка
  }
//---------------------------------------------------------
 int tenantList::getAptNo(string tName) // имя присутствует
                                        // в списке?
   {
   int aptNo;
   tenant dummy(tName, 0);
   iter = setPtrsTens.begin();
   while( iter != setPtrsTens.end() )
      {
      aptNo = (*iter)->getAptNumber(); // поиск жильца
      if(dummy == **iter++)            // в списке?
         return aptNo;                 // да
      }
   return -1;                          // нет
    }
//---------------------------------------------------------
 void tenantList::display()      // вывод списка жильцов
   {
   cout << "\nApt#\tИмя жильца\n-------------------\n";
   if( setPtrsTens.empty() )
     cout << "***Нет жильцов***\n";
   else
     {
     iter = setPtrsTens.begin();
     while( iter != setPtrsTens.end() )
        cout << **iter++;
     }
   }  // end display()
//---------------------------------------------------------

/////////////////методы класса rentRow////////////////////
rentRow::rentRow(int an) : aptNo(an)   //1-арг. конструктор
  { fill( &rent[0]. &rent[12]. 0); }
```

```
//------------------------------------------------------
void rentRow::setRent(int m, float am)
  { rent[m] = am; }
//------------------------------------------------------
float rentRow::getSumOfRow()    // сумма арендных платежей
                                // в строке
  { return accumulate( &rent[0], &rent[12], 0); }
//------------------------------------------------------
bool operator < (const rentRow& t1, const rentRow& t2)
  { return t1.aptNo < t2.aptNo; }
//------------------------------------------------------
bool operator == (const rentRow& t1, const rentRow& t2)
  { return t1.aptNo == t2.aptNo; }
//------------------------------------------------------
ostream& operator << (ostream& s, const rentRow& an)
  {
  s << an.aptNo << '\t';        // вывести номер комнаты
  for(int j=0; j<12; j++)       // вывести 12 арендных
                                // платежей
    {
    if(an.rent[j] == 0)
      s << "  0  ";
    else
      s << an.rent[j] << "  ";
    }
  s << endl;
  return s;
  }
////////////////////////////////////////////////////////
bool compareRows::operator () (rentRow* ptrR1,
                               rentRow* ptrR2) const
  { return *ptrR1 < *ptrR2; }

////////////////методы класса rentRecord////////////////
rentRecord::~rentRecord()       // деструктор
  {
  while( !setPtrsRR.empty() )   // удалить строки
                                // с платежами,
     {                 // удалить указатели из множества
     iter = setPtrsRR.begin();
     delete *iter;
     setPtrsRR.erase(iter);
     }
  }
//------------------------------------------------------
void rentRecord::insertRent(int aptNo, int month, float amount)
  {
  rentRow searchRow(aptNo);     // временная строка
                                // с тем же aptNo
  iter = setPtrsRR.begin();     // поиск setPtrsRR
  while( iter != setPtrsRR.end() )
    {
    if(searchRow==**iter)              // rentRow найден?
      {                                // да, заносим
      (*iter)->setRent(month, amount); // строку в
                                       // список
       return;
       }
```

продолжение

Листинг 16.3 (*продолжение*)

```
    else
       iter++;
    }                               // не найден
  rentRow* ptrRow = new rentRow(aptNo);// новая строка
  ptrRow->setRent(month, amount);  // занести в нее платеж
  setPtrsRR.insert(ptrRow);        // занести строку в
                                    // вектор
  } // конец insertRent()
//---------------------------------------------------------
void rentRecord::display()
  {
  cout << "\nAptNo\tЯнв  Фев  Мар  Апр  Май  Июн  "
       <<          "Июл  Авг  Сен  Окт  Ноя  Дек\n"
       << "---------------------------------"
       << "---------------------------------\n";
  if( setPtrsRR.empty() )
    cout << "***Нет платежей!***\n";
  else
    {
    iter = setPtrsRR.begin();
    while( iter != setPtrsRR.end() )
       cout << **iter++;
     }
  }
//---------------------------------------------------------
float rentRecord::getSumOfRents()   // сумма всех платежей
  {
  float sumRents = 0.0;
  iter = setPtrsRR.begin();
  while( iter != setPtrsRR.end() )
    {
    sumRents += (*iter)->getSumOfRow();
    iter++;
    }
  return sumRents;
  }
//---------------------------------------------------------

////////////////методы класса rentInputScreen/////////////
void rentInputScreen::getRent()
  {
  cout << "Введите имя жильца: ";
  getaLine(renterName);
  aptNo = ptrTenantList->getAptNo(renterName);
  if(aptNo > 0)                  // если имя найдено,
    {                            // получить сумму платежа
    cout << "Введите сумму платежа (345.67): ";
    cin >> rentPaid;
    cin.ignore(80, '\n');
    cout << "Введите номер месяца оплаты (1-12): ";
    cin >> month;
    cin.ignore(80, '\n');
    month--;                 // (внутренняя нумерация 0-11)
    ptrRentRecord->insertRent(aptNo, month, rentPaid);
    }
  else                       // возврат
```

```
    cout << "Такого жильца нет.\n";
  }  // end getRent()
//---------------------------------------------------------

//////////////////методы класса expense//////////////////
bool operator < (const expense& e1, const expense& e2)
  {                              // сравнивает даты
  if(e1.month == e2.month)       // если тот же месяц,
     return e1.day < e2.day;     // сравнить дни
  else                           // иначе,
     return e1.month < e2.month; // сравнить месяцы
  }
//---------------------------------------------------------
bool operator == (const expense& e1, const expense& e2)
  { return e1.month == e2.month && e1.day == e2.day; }
//---------------------------------------------------------
ostream& operator << (ostream& s, const expense& exp)
  {
  s << exp.month << '/' << exp.day << '\t' << exp.payee << '\t'
  s << exp.amount << '\t' << exp.category << endl;
  return s;
  }
//---------------------------------------------------------

/////////////////////////////////////////////////////////
bool compareDates::operator () (expense* ptrE1,
                                expense* ptrE2) const
  { return *ptrE1 < *ptrE2; }
//---------------------------------------------------------

/////////////////////////////////////////////////////////
bool compareCategories::operator () (expense* ptrE1,
                                     expense* ptrE2) const
  { return ptrE1->category < ptrE2->category; }
//---------------------------------------------------------

/////////////////методы класса expenseRecord/////////////
expenseRecord::~expenseRecord()         // деструктор
  {
  while( !vectPtrsExpenses.empty() )   // удалить объекты
                                       // expense
     {                                 // удалить
                                       // указатели на вектор
    iter = vectPtrsExpenses.begin();
    delete *iter;
    vectPtrsExpenses.erase(iter);
    }
  }
//---------------------------------------------------------
void expenseRecord::insertExp(expense* ptrExp)
  { vectPtrsExpenses.push_back(ptrExp); }
//---------------------------------------------------------
void expenseRecord::display()
  {
  cout << "\nДата\tПолучатель\t\tСумма\tКатегория\n"
       << "----------------------------------------\n";
```

продолжение

Листинг 16.3 (*продолжение*)

```
  if( vectPtrsExpenses.size() == 0 )
    cout << "***Расходов нет***\n";
  else
    {
    sort( vectPtrsExpenses.begin(), // сортировка по дате
          vectPtrsExpenses.end(), compareDates() );
      iter = vectPtrsExpenses.begin();
      while( iter != vectPtrsExpenses.end() )
       cout << **iter++;
    }
  }
//---------------------------------------------------------
float expenseRecord::displaySummary() // используется при
                                      // составлении
                                      // годового отчета
  {
  float totalExpenses = 0;            // сумма, все
                                      // категории

  if( vectPtrsExpenses.size() == 0 )
    {
    cout << "\tВсе категории\t0\n";
    return 0;
    }
  // сортировать по категории
  sort( vectPtrsExpenses.begin(),
      vectPtrsExpenses.end(), compareCategories() );

  // по каждой категории сумма записей

  iter = vectPtrsExpenses.begin();
  string tempCat = (*iter)->category;
  float sumCat = 0.0;
  while( iter != vectPtrsExpenses.end() )
    {
    if(tempCat == (*iter)->category)
      sumCat += (*iter)->amount; // та же категория
    else
      {                          // другая
      cout << '\t' << tempCat << '\t' << sumCat << endl;
      totalExpenses += sumCat;   // прибавить предыдущую
                                 // категорию
        tempCat = (*iter)->category;
        sumCat = (*iter)->amount;// прибавить конечное
                                 // значение суммы
      }
    iter++;
    } // end while
  totalExpenses += sumCat;          // прибавить сумму
                                    // конечной
                                    // категории
  cout << '\t' << tempCat << '\t' << sumCat << endl;
  return totalExpenses;
  } // конец displaySummary()
  //---------------------------------------------------------
```

```
//////////////методы класса expenseInputScreen/////////////
expenseInputScreen::expenseInputScreen(expenseRecord* per) : ptrExpenseRecord(per)
  { /*пусто*/ }
  //---------------------------------------------------
void expenseInputScreen::getExpense()
  {
  int month, day;
  string category, payee;
  float amount;

  cout << "Введите месяц (1-12): ";
  cin >> month;
  cin.ignore(80, '\n');
  cout << "Введите день (1-31): ";
  cin  >> day;
  cin.ignore(80, '\n');
  cout << "Введите категорию расходов (Ремонт, Налоги): ";
  getaLine(category);
  cout << "Введите получателя "
      << "(ПростоквашиноЭлектроСбыт): ";
  getaLine(payee);
  cout << "Введите сумму (39.95): ";
  cin >> amount;
  cin.ignore(80, '\n');
  expense* ptrExpense = new
           expense(month, day, category, payee, amount);
  ptrExpenseRecord->insertExp(ptrExpense);
  }
//---------------------------------------------------

////////////////методы класса annualReport////////////////
annualReport::annualReport(rentRecord* pRR,
       expenseRecord* pER) :    ptrRR(pRR), ptrER(pER)
  { /* пусто */ }
//---------------------------------------------------
void annualReport::display()
  {
  cout << "Годовой отчет\n-------------\n";
  cout << "Доходы\n";
  cout << "\tАрендная плата\t\t";
  rents = ptrRR->getSumOfRents();
  cout << rents << endl;

  cout << "Расходы\n";
  expenses = ptrER->displaySummary();
  cout << "\nБаланс\t\t\t" << rents - expenses << endl;
  }
//---------------------------------------------------

////////////////методы класса userInterface//////////////
userInterface::userInterface()
  {
  //это жизненно важно для программы
  ptrTenantList      = new tenantList;
  ptrRentRecord      = new rentRecord;
  ptrExpenseRecord   = new expenseRecord;
  }
```

продолжение

Листинг 16.3 (*продолжение*)

```
//--------------------------------------------------------------
userInterface::~userInterface()
  {
  delete ptrTenantList;
  delete ptrRentRecord;
  delete ptrExpenseRecord;
  }
//--------------------------------------------------------------
void userInterface::interact()
  {
  while(true)
    {
    cout << "Для ввода данных нажмите 'i', \n"
         << "     'd' для вывода отчета, \n"
         << "     'q' для выхода: ";
    ch = getaChar();
    if(ch=='i')                    // ввод данных
      {
      cout << " 't' для добавления жильца, \n"
           << "   'r' для записи арендной платы, \n"
           << "   'e' для записи расходов: ";
      ch = getaChar();
      switch(ch)
        {
        //экраны ввода существуют только во время их
        //использования
        case 't': ptrTenantInputScreen =
          new tenantInputScreen(ptrTenantList);
          ptrTenantInputScreen->getTenant();
          delete ptrTenantInputScreen;
          break;
        case 'r': ptrRentInputScreen =
          new rentInputScreen(ptrTenantList, ptrRentRecord);
          ptrRentInputScreen->getRent();
          delete ptrRentInputScreen;
          break;
        case 'e': ptrExpenseInputScreen =
          new expenseInputScreen(ptrExpenseRecord);
          ptrExpenseInputScreen->getExpense();
          delete ptrExpenseInputScreen;
          break;
        default: cout << "Неизвестная функция\n";
          break;
        }                       // конец секции switch
      }                         // конец условия if
    else if(ch=='d')            // вывод данных
      {
      cout << " 't' для вывода жильцов, \n"
           << "   'r' для вывода арендной платы\n"
           << "   'e' для вывода расходов, \n"
           << "   'a' для вывода годового отчета: ";
      ch = getaChar();
      switch(ch)
        {
        case 't': ptrTenantList->display();
          break;
        case 'r': ptrRentRecord->display();
```

```
          break;
        case 'e': ptrExpenseRecord->display();
          break;
        case 'a':
          ptrAnnualReport = new annualReport(ptrRentRecord, ptrExpenseRecord);
          ptrAnnualReport->display();
          delete ptrAnnualReport;
          break;
        default: cout << "Неизвестная функция вывода\n";
          break;
        }                     // конец switch
      }                       // конец elseif
    else if(ch=='q')
      return;                 // выход
    else
      cout << "Неизвестная функция. Нажимайте только 'i', 'd' или 'q'\n";
    }                         // конец while
  }                           // конец interact()
/////////////////конец файлаlandlord.cpp/////////////////
```

Вынужденные упрощения

Код программы получился довольно длинным и все же содержащим большое количество допущений и упрощений. Текстовый пользовательский интерфейс вместо меню, окошек и современной графической оболочки; почти полное отсутствие проверок на ошибки ввода; поддержка данных только за один год.

Взаимодействие с программой

Мы прошли огонь и воду — спланировали и написали программу. Теперь интересно было бы пройти и медные трубы — посмотреть нашу программу «на ходу». Вот подходит к компьютеру Степан Печкин и нажимает «i», а затем «t» для ввода нового жильца. После соответствующих запросов программы (в скобках в конце запроса обычно пишут формат данных) он вводит информацию о жильце.

```
Нажмите 'i' для ввода данных,
'd' для вывода отчета
'q' для выхода: i
Нажмите 't' для добавления жильца
'r' для записи арендной платы
'e' для записи расходов: t
Введите имя жильца (Дядя Федор): Кот Матроскин
Введите номер комнаты: 101
```

После ввода всех жильцов домовладелец пожелал просмотреть их полный список (для краткости ограничимся пятью жильцами из двенадцати):

```
Нажмите 'i' для ввода данных,
'd' для вывода отчета
'q' для выхода: d
Нажмите 't' для вывода жильцов
'r' для вывода арендной платы
```

```
'e' для вывода расходов
'a' для вывода годового отчета: t

Apt #Имя жильца
----------------
101 Кот Матроскин
102 Пес Шарик
103 Дядя Федор
104 Корова Мурка
201 Птица Говорун
```

Для фиксации внесенной арендной платы домовладелец Степан Печкин, племянник знаменитого почтальона Игоря Ивановича Печкина из деревни Простоквашино Вознесенского района Московской области, нажимает вначале «i», затем «r» (далее мы не будем повторять пункты меню в примерах работы программы). Дальнейшее взаимодействие с программой протекает следующим образом:

```
Введите имя жильца: Пес Шарик
Введите внесенную сумму (345.67): 595
Введите месяц оплаты (1-12): 5
```

Пес Шарик послал домовладельцу чек оплаты за май в размере $595. (Имя жильца должно быть напечатано так же, как оно появлялось в списке жильцов. Более умная программа, возможно, дала бы более гибкое решение этого вопроса.)

Чтобы увидеть всю таблицу доходов от аренды помещений, нужно нажать «d», а затем «r». Вот каково состояние таблицы после внесения майской арендной платы:

```
Apt No Янв Фев Мар Апр Май Июн Июл Авг Сен Окт Ноя Дек
-----------------------------------------------------
101    695 695 695 695 695 0   0   0   0   0   0   0
102    595 595 595 595 595 0   0   0   0   0   0   0
103    810 810 825 825 825 0   0   0   0   0   0   0
104    645 645 645 645 645 0   0   0   0   0   0   0
201    720 720 720 720 720 0   0   0   0   0   0   0
Обратите внимание, оплата для дяди Федора с марта была увеличена.
Чтобы ввести значения расходов, нужно нажать «i» и «e». Например:
Введите месяц: 1
Введите день: 15
Введите категорию расходов (Ремонт, Налоги): Коммунальные услуги
Введите получателя (ПростоквашиноЭлектроСбыт): ПЭС
Введите сумму платежа: 427.23
```

Для вывода на экран таблицы расходов необходимо нажать «d» и «e ». Ниже показано начало такой таблицы.

```
Дата Получатель       Сумма   Категория
---------------------------------- 714
--------------
1/3  УльтраМегаБанк  5187.30  Закладная
1/8  ПростоВодоканал 963.0    Коммунальные услуги
1/9  СуперСтрах      4840.00  Страховка
1/15 ПЭС             727.23   Коммунальные услуги
1/22 Хлам дяди Сэма  54.81    Снабжение
1/25 Подвал Мастерз  150.00   Ремонт
2/3  УльтраМегаБанк  51В7.30  Закладная
```

Наконец, для вывода годового отчета пользователь должен нажать «d» и «e». Посмотрим на отчет за первые пять месяцев года:

```
Годовой отчет
-------------
Доходы
Арендная плата42610.12
Расходы
Закладная25936.57
Коммунальные услуги7636.15
Реклама95.10
Ремонт1554.90
Снабжение887.22
Страховка4840.00
Баланс1660.18
```

Категории расходов сортируются в алфавитном порядке. В реальной ситуации может быть довольно много бюджетных категорий, включая одни налоги, другие, третьи, расходы на поездки, ландшафтный дизайн дворовой территории, уборку помещений и т. д.

Заключение

Процесс разработки реальных проектов может проходить вовсе не так гладко, как в нашем примере. Может понадобиться не одна, а несколько итераций каждого из показанных этапов. Программисты могут по-разному представлять себе нужды пользователей, что потребует возвращения с середины этапа построения на этап развития. Пользователи тоже могут запросто изменить свои требования, не очень задумываясь о том, какие неудобства они тем самым создают для разработчиков.

Резюме

Для простых программ при их разработке может быть достаточно метода проб и ошибок. Но при разработке крупных проектов требуется более организованный подход. В этой главе мы обсудили один из возможных методов. Унифицированный процесс состоит из следующих этапов: начало, развитие, построение и внедрение. Этап развития соответствует программному анализу, а построение — планированию структуры программы и написанию кода.

В Унифицированном процессе используется прецедентный подход к разработке. Тщательно изучаются потенциальные пользователи и их требования. Диаграмма вариантов использования UML демонстрирует действующие субъекты и инициируемые ими операции (варианты использования). Любое существительное из описаний вариантов использования может в будущем стать именем класса или атрибута. Глаголы превращаются в методы.

В дополнение к диаграммам вариантов использования существует еще множество других UML-диаграмм, помогающих в более полной мере достичь взаимопо-

нимания между пользователями и разработчиками. Отношения между классами показываются на диаграммах классов, управляющие потоки — на диаграммах действий, а диаграммы последовательностей отображают взаимосвязи между объектами при выполнении вариантов использования.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Истинно ли утверждение о том, что прецедентный подход связан, прежде всего, с определением используемых в классе методов?
2. Варианты использования (кроме всего прочего) нужны для:
   а) получения сведений о проблемах, возникших в программном коде;
   б) того, чтобы узнать, какие в классах могут быть конструкторы;
   в) обеспечения выбора подходящих атрибутов класса;
   г) определения того, какие классы необходимы в программе.
3. Вариант использования — это, на самом деле, ______________________.
4. Истинно ли утверждение о том, что после создания диаграммы вариантов использования новые варианты использования можно добавлять уже после начала написания кода программы?
5. Описание вариантов использования иногда пишется в двух __________.
6. Действующим субъектом может быть:
   а) некая система, взаимодействующая с нашей;
   б) некая программная сущность, помогающая разработчику решить конкретную проблему при кодировании;
   в) человек, взаимодействующий с разрабатываемой системой;
   г) проектировщик системы.
7. Классы могут связываться между собой с помощью ____________, ____________ или ____________________.
8. Водопадный процесс:
   а) состоит из различных этапов;
   б) никогда реально не использовался;
   в) стал непригоден в связи с нехваткой воды;
   г) может протекать только в одном направлении.
9. Истинно ли утверждение о том, что UML используется только совместно с Унифицированным процессом?
10. Классы в программе могут соответствовать:
    а) существительным в описаниях вариантов использования;
    б) вариантам использования;

в) ассоциациям в диаграммах UML;

г) именам знаменитых программистов.

11. Истинно ли утверждение о том, что невнятные, общие сущности (например, такие, как «система») описаний вариантов использования должны исключаться из кандидатов в классы?

12. Истинно ли утверждение о том, что сущности с единственным атрибутом и не имеющие методов являются хорошими кандидатами в классы?

13. Что может происходить время от времени в Унифицированном процессе:

а) диаграмма вариантов использования рисуется до того, как становятся известны все варианты использования;

б) диаграмма классов рисуется до того, как написаны некоторые описания вариантов использования;

в) часть кода может быть написана до окончания работы над диаграммой классов;

г) заголовочный файл с объявлениями классов изменяется одновременно с написанием тел методов.

14. Действующие субъекты — это ____________ или ______________, взаимодействующие с ________________________.

15. Контейнерные классы STL в программе LANDLORD:

а) нельзя использовать, так как их невозможно отобразить на диаграмме вариантов использования;

б) являются подходящим местом для хранения данных о расходах;

в) нельзя использовать, так как C++ является объектно-ориентированным языком;

г) являются подходящим местом для хранения скрытых тел методов.

16. Определения методов:

а) следует помещать в заголовочные файлы;

б) не следует помещать в заголовочные файлы;

в) вероятно, не должны предоставляться пользователям;

г) обычно предоставляются пользователям.

17. Истинно ли утверждение о том, что атрибуция является одним из основных видов взаимосвязи между классами?

18. Пусть имеется ассоциативная связь между классами A и B. Пусть objA — объект класса A, а objB — объект класса B. Тогда:

а) objA может послать сообщение objB;

б) класс B должен быть подклассом класса A или наоборот;

в) objB должен быть атрибутом класса A или наоборот;

г) objB может помочь objA выполнить задачу.

19. В программе LANDLORD используется
   а) обобщение;
   б) ассоциация;
   в) интерсубординация;
   г) агрегация.
20. Истинно ли утверждение о том, что на диаграмме классов ассоциация показана в виде отношения между объектами?
21. В диаграмме последовательностей:
   а) время идет слева направо;
   б) ассоциации идут справа налево;
   в) горизонтальные стрелки представляют собой сообщения;
   г) вертикальные пунктирные линии представляют собой линию жизни.
22. Диаграмма последовательностей показывает сообщения одних __________ другим.
23. Истинно ли утверждение о том, что диаграмма последовательностей часто описывает только один вариант использования?
24. При создании нового класса на диаграмме последовательностей:
   а) рисуется прямоугольник с его именем на соответствующей высоте;
   б) большой буквой X отмечается этот момент времени;
   в) с этого места начинается блок активности;
   г) начинается его линия жизни.

Проекты

Нам, к сожалению, не хватило места на упражнения по теме данной главы. Мы лишь дадим вам на выбор несколько идей проектов, которые вы уже в состоянии разработать самостоятельно.

1. Перечитайте пояснения к программе HORSE из главы 10 «Указатели», но не подсматривайте в ее исходный код. Создайте диаграмму вариантов использования и диаграмму классов для этой программы. Результат этих действий используйте при написании заголовочного файла. Сравните то, что получилось, с примером из книги. Должно быть довольно много совпадений.

2. Перечитайте пояснения к программе ELEV из главы 13 «Многофайловые программы», но не подсматривайте в ее исходный код. Создайте диаграмму вариантов использования и диаграмму классов для этой программы. Результат этих действий используйте при написании заголовочного файла. Сравните то, что получилось, с примером из книги.

3. Создайте диаграмму вариантов использования и диаграмму классов для той сферы деятельности, которая вам наиболее близка, будь то торговля кроликами, юридическая консультация или продажа редких книг.
4. Создайте диаграмму вариантов использования и диаграмму классов для программы, которую вы всегда мечтали написать, но боялись спросить как. Если вам не хватает воображения, напишите простенький текстовый процессор типа MS Word'2000, игрушку «шахматы» или программу построения своего генеалогического древа.

Приложение A
Таблица ASCII

Таблица A. Коды символов IBM

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в C
0	00	(NULL)	Ctrl 2	—
1	01	☺	Ctrl A	—
2	02	●	Ctrl B	—
3	03	♥	Ctrl C	—
4	04	♦	Ctrl D	—
5	05	♣	Ctrl E	—
6	06	♠	Ctrl F	—
7	07	•	Ctrl G	Звуковой сигнал
8	08	◘	Backspace	Забой
9	09	○	Tab	Табуляция
10	0A	■	Ctrl J	Новая строка
11	0B	♂	Ctrl K	Верт. табуляция
12	0C	♀	Ctrl L	Прогон страницы
13	0D	♪	Enter	Возврат каретки
14	0E	♫	Ctrl N	—
15	0F	☼	Ctrl O	—
16	10	►	Ctrl P	—
17	11	◄	Ctrl Q	—
18	12	↕	Ctrl R	—
19	13	‼	Ctrl S	—
20	14	¶	Ctrl T	—
21	15	§	Ctrl U	—
22	16	▬	Ctrl V	—
23	17	↨	Ctrl W	—
24	18	↑	Ctrl X	—
25	19	↓	Ctrl Y	—
26	1A	→	Ctrl Z	—
27	1B	←	Esc	—
28	1C	∟	Ctrl \	—
29	1D	↔	Ctrl]	—
30	1E	▲	Ctrl 6	—
31	1F	▼	Ctrl -	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в C
32	20		пробел	—
33	21	!	!	—
34	22	"	"	—
35	23	#	#	—
36	24	$	$	—
37	25	%	%	—
38	26	&	&	—
39	27	'	'	—
40	28	(	(	—
41	29	)	)	—
42	2A	*	*	—
43	2B	+	+	—
44	2C	,	,	—
45	2D	-	-	—
46	2E	.	.	—
47	2F	/	/	—
48	30	0	0	—
49	31	1	1	—
50	32	2	2	—
51	33	3	3	—
52	34	4	4	—
53	35	5	5	—
54	36	6	6	—
55	37	7	7	—
56	38	8	8	—
57	39	9	9	—
58	3A	:	:	—
59	3B	;	;	—
60	3C	<	<	—
61	3D	=	=	—
62	3E	>	>	—
63	3F	?	?	—
64	40	@	@	—
65	41	A	A	—
66	42	B	B	—
67	43	C	C	—
68	44	D	D	—
69	45	E	E	—
70	46	F	F	—
71	47	G	G	—
72	48	H	H	—
73	49	I	I	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в С
74	4A	J	J	—
75	4B	K	K	—
76	4C	L	L	—
77	4D	M	M	—
78	4E	N	N	—
79	4F	O	O	—
80	50	P	P	—
81	51	Q	Q	—
82	52	R	R	—
83	53	S	S	—
84	54	T	T	—
85	55	U	U	—
86	56	V	V	—
[illegible]	57	W	W	—
88	58	X	X	—
89	59	Y	Y	—
90	5A	Z	Z	—
91	5B	[	[	—
92	5C	\	\	—
93	5D	]	]	—
94	5E	^	^	—
95	5F	_	_	—
96	60	`	`	—
97	61	a	a	—
98	62	b	b	—
99	63	c	c	—
100	64	d	d	—
101	65	e	e	—
102	66	f	f	—
103	67	g	g	—
104	68	h	h	—
105	69	i	i	—
106	6A	j	j	—
107	6B	k	k	—
108	6C	l	l	—
109	6D	m	m	—
110	6E	n	n	—
111	6F	o	o	—
112	70	p	p	—
113	71	q	q	—
114	72	r	r	—
115	73	s	s	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в С
116	74	t	t	—
117	75	u	u	—
118	76	v	v	—
119	77	w	w	—
120	78	x	x	—
121	79	y	y	—
122	7A	z	z	—
123	7B	{	{	—
124	7C	|	|	—
125	7D	}	}	—
126	7E	~	~	—
127	7F	Δ	Ctrl <	—
128	80	Ä	Alt 128	—
129	81	ü	Alt 129	—
130	82	é	Alt 130	—
131	83	É	Alt 131	—
132	84	ä	Alt 132	—
133	85	à	Alt 133	—
134	86	å	Alt 134	—
135	87	ç	Alt 135	—
136	88	ê	Alt 136	—
137	89	ë	Alt 137	—
138	8A	è	Alt 138	—
139	8B	ï	Alt 139	—
140	8C	î	Alt 140	—
141	8D	ì	Alt 141	—
142	8E	Ä	Alt 142	—
143	8F	Å	Alt 143	—
144	90	É	Alt 144	—
145	91	æ	Alt 145	—
146	92	Æ	Alt 146	—
147	93	ô	Alt 147	—
148	94	ö	Alt 148	—
149	95	ò	Alt 149	—
150	96	û	Alt 150	—
151	97	ù	Alt 151	—
152	98	ÿ	Alt 152	—
153	99	Ö	Alt 153	—
154	9A	Ü	Alt 154	—
155	9B	õ	Alt 155	—
156	9C	£	Alt 156	—
157	9D	¥	Alt 157	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в C
158	9E	û	Alt 158	—
159	9F	ü	Alt 159	—
160	A0	á	Alt 160	—
161	A1	í	Alt 161	—
162	A2	ó	Alt 162	—
163	A3	ú	Alt 163	—
164	A4	ñ	Alt 164	—
165	A5	Ñ	Alt 165	—
166	A6	a	Alt 166	—
167	A7	o	Alt 167	—
168	A8	¿	Alt 168	—
169	A9	⌐	Alt 169	—
170	AA	¬	Alt 170	—
171	AB	½	Alt 171	—
172	AC	¼	Alt 172	—
173	AD	¡	Alt 173	—
174	AE	«	Alt 174	—
175	AF	»	Alt 175	—
176	B0	░	Alt 176	—
177	B1	▒	Alt 177	—
178	B2	▓	Alt 178	—
179	B3	│	Alt 179	—
180	B4	┤	Alt 180	—
181	B5	╡	Alt 181	—
182	B6	╢	Alt 182	—
183	B7	╖	Alt 183	—
184	B8	╕	Alt 184	—
185	B9	╣	Alt 185	—
186	BA	║	Alt 186	—
187	BB	╗	Alt 187	—
188	BC	╝	Alt 188	—
189	BD	╜	Alt 189	—
190	BE	╛	Alt 190	—
191	BF	┐	Alt 191	—
192	C0	└	Alt 192	—
193	C1	┴	Alt 193	—
194	C2	┬	Alt 194	—
195	C3	├	Alt 195	—
196	C4	─	Alt 196	—
197	C5	┼	Alt 197	—
198	C6	╞	Alt 198	—
199	C7	╟	Alt 199	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в С
200	C8	╚	Alt 200	—
201	C9	╔	Alt 201	—
202	CA	╩	Alt 202	—
203	CB	╦	Alt 203	—
204	CC	╠	Alt 204	—
205	CD	═	Alt 205	—
206	CE	╬	Alt 206	—
207	CF	╧	Alt 207	—
208	D0	╨	Alt 208	—
209	D1	╤	Alt 209	—
210	D2	╥	Alt 210	—
211	D3	╙	Alt 211	—
212	D4	╘	Alt 212	—
213	D5	╒	Alt 213	—
214	D6	╓	Alt 214	—
215	D7	╫	Alt 215	—
216	D8	╪	Alt 216	—
217	D9	┘	Alt 217	—
218	DA	┌	Alt 218	—
219	DB	█	Alt 219	—
220	DC	▄	Alt 220	—
221	DD	▌	Alt 221	—
222	DE	▐	Alt 222	—
223	DF	▀	Alt 223	—
224	E0	α	Alt 224	—
225	E1	β	Alt 225	—
226	E2	Γ	Alt 226	—
227	E3	π	Alt 227	—
228	E4	Σ	Alt 228	—
229	E5	σ	Alt 229	—
230	E6	μ	Alt 230	—
231	E7	τ	Alt 231	—
232	E8	Φ	Alt 232	—
233	E9	Θ	Alt 233	—
234	EA	Ω	Alt 234	—
235	EB	δ	Alt 235	—
236	EC	∞	Alt 236	—
237	ED	φ	Alt 237	—
238	EE	∈	Alt 238	—
239	EF	∩	Alt 239	—
240	F0	≡	Alt 240	—
241	F1	±	Alt 241	—

Десятичный код	Шестнадцатеричный код	Символ	Сочетание клавиш	Использование в C
242	F2	≥	Alt 242	—
243	F3	≤	Alt 243	—
244	F4	⌠	Alt 244	—
245	F5	⌡	Alt 245	—
246	F6	÷	Alt 246	—
247	F7	≈	Alt 247	—
248	F8	°	Alt 248	—
249	F9	•	Alt 249	—
250	FA	·	Alt 250	—
251	FB	√	Alt 251	—
252	FC	η	Alt 252	—
253	FD	^{2}	Alt 253	—
254	FE	■	Alt 254	—
255	FF	(пустой)	Alt 255	—

Те сочетания клавиш, в которых встречается Ctrl, следует вводить таким образом: удерживать клавишу Ctrl и, не отпуская ее, нажать вторую клавишу из таблицы. Эти последовательности клавиш предназначены только для клавиатур персональных компьютеров (PC). Для других клавиатур могут назначаться другие сочетания.

Символы расширенной таблицы кодов ASCII вводятся с помощью нажатия клавиши Alt с последующим набором десятичного кода на клавиатуре.

Приложение Б
Таблица приоритетов операций C++

Таблица приоритетов операций

Эта таблица является более полной версией такой же таблицы из главы 3. Она включает в себя поразрядные операции << и >>, которые, хотя это и не описано в нашей книге, могут быть перегружены для потокового ввода/вывода.

Таблица Б. Таблица приоритета операций

Тип операторов	Операторы
Контекст	::
Разное	[], (), . (точка), ->, постфиксы ++ и --
Унарные	Префиксы ++ и --, &, *, +, -, !
Арифметические	Умножение *, /, %, сложение +, -
Поразрядный сдвиг	<<, >>
Относительные (сравнение)	Неравенства <, >, <=, >=
	Равенства ==, !=
Поразрядные логические	&, ^, |
Логические	&&, ||
Условные	?:
Присваивания	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
Последовательность	, (запятая)

Зарезервированные слова

Ключевые (зарезервированные) слова реализуют различные специфические функции C++. Их нельзя использовать в качестве имен переменных или элементов программ. Многие из них являются общими для C и C++, но есть некоторые, характерные только для C++. Некоторые компиляторы могут поддерживать дополнительные ключевые слова, которые обычно начинаются с одного или двух символов подчеркивания, например, _cdecl или __int16.

```
asm
auto

bool
break

case
catch
char
class
const
const_cast
continue

default
delete
do
double
dynamic_cast

else
enum
explicit
export
extern

false
float
for
friend

goto

if
inline
int

long

main
mutable

namespace
new

operator

private
protected
public

register
reinterpret_cast
return

short
signed
sizeof
```

```
static
static_cast
struct
switch

template
this
true
try
typedef
typeid
typename

union
unsigned
using

virtual
void
volatile

wchar_t
while
```

Приложение В
Microsoft Visual C++

- ✦ Элементы экрана
- ✦ Однофайловые программы
- ✦ Многофайловые программы
- ✦ Программы с консольной графикой
- ✦ Отладка программ

В этом приложении рассказывается об использовании Microsoft Visual C++ (MVC++) для создания консольных приложений, например, таких, которые приводились в качестве примеров в этой книге. Разговор пойдет об MVC++ версии 6.0.

Настоящая версия MVC++ является наследником стандартного C++. Она поставляется в различных вариантах, включая недорогую студенческую комплектацию.

Мы предполагаем, что система MVC++ установлена на вашем компьютере и вы знаете, как запустить ее в Windows.

Вам понадобится, чтобы на экране отображались расширения файлов (например, .cpp), нужных для работы с MVC++. Убедитесь, что в настройках Проводника выключена функция, скрывающая расширения зарегистрированных типов файлов.

Элементы экрана

Окно MVC++ при старте системы разделено на три части. Слева — панель View (Вид). У нее две закладки: ClassView (Классы) и FileView (Файлы). Если вы откроете какой-нибудь проект, то на этой панели в закладке ClassView (Классы) будет отображаться иерархия классов программы, а в закладке FileView (Файлы) — список файлов, из которых состоит проект. При нажатии на плюсик показывается следующий уровень иерархии. Для того чтобы открыть какой-либо файл, нужно дважды щелкнуть на его имени.

Больше всего места на экране обычно занимает та часть, в которой находится текст открытого файла. Ее можно использовать в разных целях, в том числе

для отображения исходного текста программы или справки. Внизу на экране находится окошко со множеством закладок: Build (Компоновка), Debug (Отладка) и т. д. Здесь будут отображаться различные сообщения компилятора о таких выполняемых действиях, как отладка программы.

Однофайловые программы

С помощью MVC++ можно довольно легко создавать и выполнять однофайловые консольные программы.

Возможны два варианта: либо файл уже существует и его нужно править, либо файла еще нет. В любом случае начинать следует, лишь убедившись в том, что нет открытых проектов (что такое проект, мы вскоре обсудим). Нажмите на меню File (Файл). Если команда закрытия рабочей области Close Workspace недоступна, значит, открытых проектов нет и можно приступать к работе над своим. Если же она доступна, нажмите на нее для закрытия открытого в текущий момент проекта.

Компоновка существующего файла

Если исходный файл .CPP уже существует, выберите пункт Open (Открыть) из меню File (Файл). Имейте в виду, что это не то же самое, что Open WorkSpace (Открыть рабочую область). В появившемся диалоговом окне найдите нужный файл, выберите его и щелкните на кнопке Open (Открыть). Файл появится в окне документа (если в программе используется консольная графика, как в примере CIDCSTRC из главы 5 «Функции» или в примере CIRCLES из главы 6 «Объекты и классы», обратитесь к разделу «Программы создания консольного оформления» данного приложения).

Для компиляции и компоновки исходного файла выберите пункт Build (Компоновка) из одноименного меню. В диалоговом окне появится запрос на создание обычной рабочей области проекта. Ответьте утвердительно. Файл будет откомпилирован и связан со всеми необходимыми библиотечными файлами.

Для запуска программы выберите Execute (Запустить) из меню Build (Компоновка). Если все сделано правильно, появится окно с результатами работы программы.

По окончании работы программы в окне появится фраза Нажмите любую клавишу для продолжения. Ее вставляет в конец программы компилятор. Окно результатов работы должно быть видно довольно долгое время, чтобы вы могли их прочитать.

Если вы заканчиваете работу с программой, закройте ее рабочую область с помощью пункта Close Workspace (Закрыть рабочую область) из меню File (Файл). Ответьте утвердительно, когда в диалоговом окне появится запрос на закрытие всех окон с документами. Программу можно запустить и вручную, напрямую из MS DOS. Сессия MS DOS открывается в Windows из меню Start ▸ Programs (Пуск ▸ Программы) с помощью ярлыка MS-DOS Prompt (Сессия MS-DOS). В результате вы увидите черное окошко с приглашением MS DOS. Перемещайтесь по каталогам, используя команду cd <имя каталога>. Исполняемые (.EXE) файлы

программ, созданных компилятором MVC++, располагаются в подкаталоге DEBUG каталога, в котором хранятся файлы проектов. Для того чтобы запустить любые откомпилированные MVC++ программы, в том числе те, тексты которых имеются в книге, убедитесь, что вы находитесь в том же каталоге, что и файл .EXE, и наберите имя программы. Более подробную информацию можно получить, воспользовавшись Справочной системой Windows.

Создание нового файла

Чтобы создать свой файл .cpp, закройте рабочую область открытого проекта, выберите пункт New (Новый) из меню File (Файл). Щелкните на закладке Files (Файлы). Выберите нужный тип файла C++ Source File (Исходный файл C++), наберите имя файла, убедившись, что находитесь в нужном каталоге. Щелкните на кнопке OK. Появится чистое окно документа. В нем можно набирать код программы. Не забудьте сохранить файл, выбрав пункт Save As... (Сохранить как...) из меню File (Файл). Затем, как уже было описано выше, откомпилируйте свою программу.

Ошибки

Если в программе имеются ошибки, сообщения об этом будут появляться в окне Build (Компоновка) внизу экрана. Если дважды щелкнуть на строке с какой-либо ошибкой, появится стрелка в окне документа, указывающая на ту строку в коде, где эта ошибка произошла. Кроме того, если в окне Build (Компоновка) поставить курсор на строку с кодом ошибки (например, C2143) и нажать F1, то в окне документа появятся комментарии к ошибке. Ошибки нужно устранять до тех пор, пока не появится сообщение 0 error(s), 0 warning(s) (0 ошибок, 0 предупреждений). Для запуска программы выберите пункт Execute Build (Компоновка) из меню Build (Компоновка).

Одной из распространенных ошибок является отсутствие в коде программы строки

```
using namespace std:
```

Если забыть включить ее в программу, компилятор сообщит, что он не знает, что такое cout, <<, endl и т. п.

Перед началом работы над новой программой не забывайте закрывать отрытые проекты. Это гарантирует, что вы действительно начнете создавать новый проект. Чтобы открыть уже скомпонованную программу, выберите Open Workspace (Открыть рабочую область) из меню File (Файл). В диалоговом окне перейдите в нужную директорию и дважды щелкните на файле с расширением .DSW.

Информация о типах в процессе исполнения (RTTI)

Некоторые программы, такие, как EMPL_IO.CPP из главы 12 «Потоки и файлы», используют RTTI. Можно настроить MVC++ так, чтобы эта функция работала.

Выберите пункт Settings (Настройки) из меню Project (Проект) и щелкните на закладке C/C++. Из списка категорий выберите C++ Language (Язык C++). Установите флажок Enable Run-Time Type Information (Включить RTTI). Затем нажмите OK. Эта функция системы MVC++ позволяет избежать появления многих ошибок компилятора и компоновщика, некоторые из которых вообще только вводят в заблуждение.

Многофайловые программы

Мы продемонстрировали, как можно быстро, но непрофессионально создавать программы. Этот способ, может быть, и хорош для однофайловых программ. Но когда в проекте больше одного файла, все сразу становится значительно сложнее. Начнем с терминологии. Разберемся с понятиями *рабочая область* и *проект*.

Проекты и рабочие области

В MVC++ используется концепция рабочих областей, что на один уровень абстракции выше, чем проект. Дело в том, что в рабочей области может содержаться несколько проектов. Она состоит из каталога и нескольких конфигурационных файлов. Внутри нее каждый проект может иметь собственный каталог или же файлы всех проектов могут храниться в одном каталоге рабочей области.

Наверное, концептуально проще предположить, что у каждого проекта имеется своя отдельная рабочая область. Именно это мы и будем предполагать при дальнейшем обсуждении.

Проект соответствует приложению (программе), которое вы разрабатываете. Он состоит из всех файлов, нужных программе, а также из информации об этих файлах и их компоновке. Результатом всего проекта обычно является один исполняемый .EXE файл (возможны и другие результаты, например файлы .DLL).

Работа над проектом

Предположим, что файлы, которые вы собираетесь включить в свой проект, уже существуют и находятся в известном каталоге. Выберите пункт New (Новый) из меню File (Файл) и щелкните на закладку Projects (Проекты) в появившемся диалоговом окне. Выберите из списка Win32 Console Application. В поле Location (Размещение) выберите путь к каталогу, но само имя каталога *не указывайте*. Затем наберите имя каталога, содержащего файлы, в поле Project Name (Имя проекта) (щелкнув на кнопке, находящейся справа от поля Location (Размещение), вы сможете выбрать нужный каталог, переходя по дереву, но убедитесь, что в этот путь не включено само имя каталога). Убедитесь, что флажок Create New Workspace (Создать новую рабочую область) установлен, и нажмите OK.

Например, если файлы находятся в каталоге c:\Book\Ch13\Elev, следует в поле Location (Размещение) указать c:\Book\Ch13\, а в поле Project Name Field (Имя проекта) — Elev. Имя проекта автоматически добавляется к пути (если бы в по-

ле Location (Размещение) был указан полный путь, то результирующий путь к файлам был бы c:\Book\Ch13\Elev\Elev, а это, конечно, совсем не то, что нам нужно). После этого появляется еще одно диалоговое окно. Следует убедиться, что выбрана кнопка An Empty Project (Пустой Проект), и щелкнуть на Finish (Закончить). В следующем окне нажмите OK.

В это время в указанном каталоге создаются различные файлы, относящиеся к проекту. Они имеют расширения .DSP, .DSW и т. п. Кроме того, создается пустая папка DEBUG, в которой будет храниться откомпилированный исполняемый файл.

Добавление исходного файла

А теперь к проекту нужно прицепить исходные файлы, включающие в себя файлы .CPP и .H, которые должны быть доступны для просмотра в закладке Файл. Выберите пункт Add to Project (Добавить к проекту) из меню Project (Проект), щелкните на Files (Файлы), выберите нужные файлы и нажмите OK. Теперь файлы включены в проект, и их можно просматривать в закладке FileView (Файлы) окна View (Вид). Щелкнув на закладке ClassView (Классы) в том же окне, можно просмотреть структуру классов, функций и атрибутов.

Чтобы открыть файл проекта для просмотра и редактирования, нужно дважды щелкнуть на его значке в закладке FileView (Файлы). Вторым способом открытия файла является его выбор с помощью команды Open (Открыть) меню File (Файл).

Определение местонахождения заголовочных файлов

В вашем проекте могут использоваться заголовочные файлы (обычно с расширением .H), например, такие, как MSOFTCON.H в программах, использующих консольную графику. Их не нужно включать в проект, если, конечно, вы не хотите просматривать их код, но компилятору нужно знать, где они находятся. Если они в том же каталоге, в котором хранятся исходные файлы, то вопросов не возникает. В противном случае нужно сообщить компилятору об их местонахождении.

Выберите пункт Options (Параметры) из меню Tools (Инструменты). Щелкните на закладке Directories (Каталоги). Для того чтобы увидеть список каталогов с включаемыми файлами компилятора, нужно вначале выбрать Include Files (Включаемые файлы) из предлагаемого списка под названием Show Directories For (Показать директории для). Дважды щелкните на поле с точечками в последней строке списка. Затем перейдите в директорию, в которой хранится нужный заголовочный файл. Точечки в поле будут заменены на новый путь к файлу. Нажмите OK. Еще можно набрать полный путь к этой директории в поле Location (Размещение).

Сохранение, закрытие и открытие проектов

Чтобы сохранить проект, выберите пункт Save Workspace (Сохранить рабочую область). Чтобы закрыть — пункт Close Workspace (Закрыть рабочую область) (отвечайте утвердительно на запрос закрытия всех окон документов). Для открытия существующего проекта выберите пункт Open Workspace (Открыть рабочую

область) из меню File (Файл). Перейдите в требуемый каталог и откройте нужный файл с расширением .DSW. Щелкните на кнопке Open (Открыть).

Компиляция и компоновка

Как и в случае однофайловых программ, простейшим способом откомпилировать, скомпоновать и запустить многофайловую программу является выбор пункта Execute (Выполнить) из меню Build (Компоновка). Если запускать программу прямо из системы не требуется, можно выбрать пункт Build (Компоновка) из одноименного меню.

Программы с консольной графикой

Программы, использующие функции консольной графики (описываемые в приложении Д), требуют нескольких дополнительных действий по сравнению с другими программами. Вам понадобятся файлы MSOFTCON.H и MSOFTCON.CPP. Они были созданы специально для этой книги, поэтому их можно найти только на сайте, адрес которого указан во введении.

- Откройте исходный файл программы, как это было описано выше. В этом файле должна быть строка #include "msoftcon.h".
- Выберите пункт Build (Компоновка) из одноименного меню. Ответьте Да на вопрос о создании обычной рабочей области проекта. Будет создан проект, но компилятор выдаст сообщение о том, что не найден файл msoftcon.h. В этом файле содержатся объявления функций консольной графики.
- Проще всего скопировать этот файл в каталог с исходными файлами. Более правильно было бы сообщить компилятору о том, где он может найти этот файл. Следуйте инструкциям раздела «Определение местонахождения заголовочных файлов» данного приложения.
- Теперь снова попробуйте скомпоновать свою программу. На этот раз компилятор найдет заголовочный файл, но будет долго морщиться от каждой найденной в нем «ошибки», поскольку компоновщик не знает, где искать определения графических функций. А они лежат в файле MSOFTCON.CPP. Подключите его к программе, как это было описано в разделе «Добавление исходных файлов» данного приложения.

Теперь программа должна откомпилироваться и скомпоноваться нормально. Выберите Execute (Выполнить) из меню Build (Компоновка) для того, чтобы посмотреть, как она работает.

Отладка программ

В главе 3 «Циклы и ветвления» мы предлагали использовать отладчик для того, чтобы легче было разобраться, что такое цикл. Теперь перед вами конкретные рекомендации по использованию встроенного отладчика Microsoft Visual C++.

Те же действия помогут понять, где именно встречается ошибка, и устранить ее. Мы будем говорить об однофайловых программах, но тот же подход применяется с небольшими вариациями для многофайловых программ.

Начните с обычной компоновки своей программы. Исправьте все ошибки компилятора и компоновщика. Убедитесь, что листинг программы доступен в окне редактирования.

Пошаговая трассировка

Чтобы запустить отладчик, просто нажмите F10. Вы увидите желтую стрелку рядом с окном документа, указывающую на строку с открывающей фигурной скобкой в main(). Если вы хотите начинать трассировку не с начала, установите курсор на нужную строку. Затем из меню Debug (Отладка), которое в этом режиме заменяет меню Build (Компоновка), выберите Start Debug (Начать отладку), а после этого — Run to Cursor (Выполнить до курсора). Желтая стрелка теперь появится следом за выбранным выражением.

Нажмите F10. Это приведет к тому, что отладчик перейдет на следующую команду, которую можно исполнять. Соответственно, туда же сдвинется стрелка. Каждое нажатие F10 означает шаг трассировки, то есть переход к следующему выражению. Если вы прогоняете цикл, то увидите, как стрелка доходит до последнего выражения, а потом перескакивает снова на начало цикла.

Просмотр переменных

Существует возможность просмотра значений переменных в процессе пошаговой трассировки. Щелкните на закладке Locals (Локальные) в левом нижнем углу экрана, чтобы увидеть локальные переменные. Закладка Auto (Авто) покажет выборку переменных, сделанную компилятором.

Если требуется создать собственный набор просматриваемых переменных, внесите их в окно просмотра, расположенное в правом нижнем углу экрана. Чтобы сделать это, щелкните правой кнопкой мыши на имени переменной в исходном коде. Из появившегося всплывающего меню выберите пункт QuickView (Быстрый просмотр). В диалоговом окне щелкните на Add watch (Добавить) для подтверждения своего намерения. Имя переменной и ее текущее значение появится в окне просмотра. Если переменная пока недоступна (например, еще не определена в программе), окно просмотра выдаст сообщение об ошибке вместо значения рядом с именем переменной.

Пошаговая трассировка функций

Если в программе используются функции, есть замечательная возможность осуществить пошаговую трассировку каждого выражения функции. Это делается с помощью клавиши F11. В отличие от режима трассировки по F10, который рассматривает каждую встреченную функцию как одно выражение, такой режим позволяет «войти» внутрь функции и выявить возможные ошибки в ней. Если

вы заставите по F11 проходить библиотечные функции, например, cout <<, то будете любоваться исходным кодом библиотеки. Это процесс долгий, поэтому советуем пользоваться этой возможностью, только если вам действительно важно знать, что происходит внутри рутинных функций. Впрочем, можно по мере необходимости чередовать нажатия F10 и F11, в зависимости от необходимости выявления ошибок во внутренних функциях.

Точки останова

Точки останова, как следует из их названия, позволяют временно прерывать работу программы в указанных местах. Когда их нужно использовать? Мы уже показали, как можно заставить отладчик прогонять программу только до курсора. Но бывают случаи, когда необходимо иметь несколько таких точек. Например, можно остановить программу в конце условия if, а также после соответствующего else. Точки останова решают эту задачу, поскольку их можно ставить в неограниченном количестве. У них есть еще и расширенные функции, которые мы здесь рассматривать не будем.

Рассмотрим, как вставить точки останова в листинги. Во-первых, установите курсор на ту строку, в которой программа при трассировке должна остановиться. Нажмите правую кнопку мыши и выберите из меню Insert/Delete Breakpoint (Вставить/Удалить точку останова). Напротив этой строки кода появится красный кружок. Теперь, даже если вы просто запустите программу на исполнение (выбрав Debug/Go (Отладка/Запуск), например), программа остановится на этой строке. Можно на этом временном срезе проверить значения переменных, произвести пошаговую трассировку кода и т. д. В общем, можно сделать все, что угодно. Хоть вообще выйти из программы.

Чтобы удалить точку останова, нажмите правую кнопку мыши и выберите пункт Удалить из появившегося меню.

Есть еще множество других функций отладчика, мы показали лишь основные, чтобы вам было с чего начать работу.

Приложение Г
Borland C++ Builder

- Запуск примеров в C++ Builder
- Очистка экрана
- Создание нового проекта
- Задание имени и сохранение проекта
- Работа с существующими файлами
- Компиляция, связывание и запуск программ
- Добавление заголовочного файла к проекту
- Проекты с несколькими исходными файлами
- Программы с консольной графикой
- Отладка

В этом приложении рассказывается о том, как использовать Borland C++ Builder для создания консольных приложений, являющихся разновидностью примеров, приведенных в этой книге.

C++ Builder — это наилучшая среда разработки, выпущенная фирмой Borland. Она очень хорошо согласуется со стандартом C++, никаких проблем с совместимостью среды и языка не возникает. Существует студенческая версия системы, стоимость которой равна примерно $100, также есть и бесплатная версия, ее можно скачать с сайта фирмы Borland. Если вы захотите ею воспользоваться, тексты программ придется писать в текстовом редакторе Notepad или каком-нибудь подобном ему. В данном приложении рассматривается C++ Builder версии 5.0.

Мы предполагаем, что система установлена на вашем компьютере и вы умеете ее запускать в Windows.

Вам понадобится, чтобы на экране отображались расширения файлов (например, .cpp), нужных для работы с MVC++. Убедитесь, что в настройках Проводника выключена функция, скрывающая расширения зарегистрированных типов файлов.

Запуск примеров в C++ Builder

Программы из нашей книги потребуют минимальных изменений для запуска из среды C++ Builder. Дадим краткие рекомендации.

Откомпилировать и запустить в окне Сессии MS-DOS все программы можно, вообще не внося никаких изменений. Тем не менее, если вы хотите запускать их из-под оболочки C++ Builder с помощью команды Run (Пуск) из одноименного меню, потребуется вставить в конец текста программы строчки, которые будут удерживать на экране окно с результатами работы программы достаточно долго. Это можно сделать за два шага:

- вставьте выражение getch(); перед последним return в секции main(). С помощью этой функции вы сможете увидеть результаты работы программы на экране;
- вставьте выражение #include <conio.h> в начало main(). Это необходимо для getch().

Если в программе используется консольная графика, потребуются дополнительные изменения, о которых будет сказано в этом приложении несколько позже.

В в этом же приложении мы расскажем более подробно об использовании C++ Builder для редактирования, компиляции, связывания и выполнения консольных программ.

Очистка экрана

При первом запуске C++ Builder показывает на экране некоторые объекты, которые вам, в принципе, не понадобятся при создании консольных программ. Справа вы увидите окно, называющееся Form1. Закройте его (щелкнув на крестике в правом верхнем углу). Также вам не понадобится и Object Inspector (Инспектор объектов), закройте и его. Вам придется избавляться от этих окошек при каждом запуске C++ Builder.

Когда вы закроете окно Form1, то обнаружите под ним еще одно с каким-то исходным кодом на C++. Это окно называется *редактором кода*. Именно здесь вы будете просматривать исходные файлы и писать свои программы. Тем не менее при запуске вы видите файл Unit1, который вас совершенно не интересует. Закройте окно и нажмите Нет, если система спросит, не хотите ли вы сохранить изменения.

C++ Builder при запуске открывает множество панелей инструментов, и часть из них вам тоже не нужна. Вам, скорее всего, понадобятся панели Стандартная (Standard) и Debug (Отладка). Уберите все остальные панели, выбрав из меню Вид (View) пункт Toolbars (Панели инструментов) и убрав галочки перед их именами.

Создание нового проекта

C++ Builder, как и другие современные компиляторы, мыслит в терминах *проектов* при создании программ. Проект состоит из одного или нескольких исходных файлов, а также из ряда дополнительных файлов, таких как файлы ресурсов и

определений. Впрочем, они нам сейчас не понадобятся. Результатом создания программы является обычно один исполняемый .EXE-файл.

Чтобы начать создавать проект, выберите пункт New... (Новый...) из меню File (Файл). Вы увидите диалоговое окно, называющееся New Items (Новые элементы). Выберите закладку New (Новые), если это необходимо. Затем щелкните дважды на значке Console Wizard (Мастер консольных приложений). Убедитесь, что в диалоговом окне в качестве Source Type (Тип кода) выбран C++ и что установлен флажок Console Application (Консольное приложение). Снимите флажки Use VCL, Multi Threaded и Specify Project Source. Нажмите OK. Ответьте отрицательно, если система спросит, сохранить ли изменения в Project1. В новом окне редактора кода вы увидите следующий исходный текст:

```
//---------------------------------------------------------

#pragma hdrstop

//---------------------------------------------------------

#pragma argsused
int main(int argc,char **argv [])
{
return 0;
}

//---------------------------------------------------------
```

Это как бы скелет будущей консольной программы. Некоторые из этих строк вам не нужны, а некоторые придется добавить. Мы сейчас проделаем необходимые изменения и добавим выражение, выводящее какой-нибудь текст, чтобы убедиться в том, что программа работает. Результат:

```
//test1.cpp
#include <iostream>
#include <conio.h>
using namespace std;
//#pragma hdrstop        //не понадобится
//#pragma argsused       // не понадобится

int main()               //аргументы не требуются
{
cout <<"Счастливы те, кто "
<<"впервые компилирует программу.";
getch();
return 0;
}
```

Две директивы #pragma оказались не нужны, также не нужны оказались аргументы main().

Если вы в таком виде запустите программу, то убедитесь, что окно с результатами слишком быстро исчезает с экрана. Это исправляется вставкой выражения getch(); в конец программы, непосредственно перед последним return. В резуль-

тате такого изменения программа будет ожидать нажатия какой-либо клавиши и только после этого уберет окно вывода результатов с экрана. Функции getch() требуется заголовочный файл conio.h, который мы включаем с помощью соответствующего выражения в начале программы.

Если вы создаете свою программу, вам понадобится этот скелет, его вы и будете изменять и дополнять. Если у вас уже имеется какой-то файл, читайте раздел «Работа с существующими файлами» данного приложения.

Задание имени и сохранение проекта

Переименовать и сохранить нужно как исходный файл, так и проект, в котором он находится. Компилятор автоматически именует исходный файл Unit1.cpp. Для того чтобы его переименовать и сохранить, выберите пункт Save As... (Сохранить как...) из меню File (Файл). Найдите нужный каталог для своего проекта, назовите как-нибудь более осмысленно свой файл (сохранив при этом расширение .CPP) и нажмите Save (Сохранить).

Информация о проекте хранится в файле с расширением .BPR. Поэтому когда вы сохраняете проект, вы на самом деле запоминаете состояние как файла (или файлов) .CPP, так и файла .BPR. При создании нового проекта ему дается автоматическое имя Project1 (или Project с какой-то другой цифрой). Чтобы сохранить проект и изменить его название, выберите пункт Save Project As... (Сохранить проект как...) из меню File (Файл). Перейдите в каталог, в котором вы хотите сохранить файл, наберите его имя с расширением .BPR и нажмите Save (Сохранить).

Работа с существующими файлами

Рассмотрим, как нужно создавать проекты при имеющемся исходном файле. Например, если вы скачаете с сайта нашего издательства примеры программ, то как раз окажетесь в такой ситуации. Мы здесь будем рассматривать только программы с одним исходным файлом.

Допустим, что файл называется MYPROG.CPP. Убедитесь, что он находится в том каталоге, в котором вы будете создавать проект. Выберите пункт Open (Открыть) из меню File (Файл). Выберите файл в диалоговом окне и нажмите Open (Открыть). Файл появится в окне редактора кода. Выскочит диалоговое окошко. В нем будет вопрос, хотите ли вы создать проект, в котором данный файл компилировался бы и запускался. Ответьте утвердительно. Проект, таким образом, создан.

Теперь нужно его переименовать и сохранить. Выберите из меню File (Файл) пункт Save Project As... (Сохранить проект как...). Замените имя PROJECT1.BPR на какое-либо другое (обычно выбирают совпадающее с именем программы): MYPROG.BPR. Нажмите Save (Сохранить). Вот и все.

Компиляция, связывание и запуск программ

Чтобы создать исполняемый файл, выберите пункт Make (Сделать) или Build (Компоновка) из меню Project (Проект). Из вашего .CPP будет создан объектный файл .OBJ, а из него, в свою очередь, — файл .EXE. Например, если вы компилируете MYPROG.CPP, результатом будет создание файла MYPROG.EXE. Если будут обнаружены ошибки компилятора или компоновщика, они будут отображены. Постарайтесь добиться их исчезновения.

Запуск программы в C++ Builder

Если вы модифицировали программу, вставив в нее выражение getch(), как было описано ранее, вы можете откомпилировать, связать и запустить программу прямо в C++ Builder буквально одной командой Run (Пуск) из одноименного меню. Если никаких ошибок не обнаружено, появится окно результатов.

Запуск программы в MS DOS

Можно запустить свою программу и из-под MS DOS. Сессия MS DOS открывается в Windows из меню Start ▸ Programs (Пуск ▸ Программы) с помощью ярлыка MS-DOS Prompt (Сессия MS-DOS). В результате вы увидите черное окошко с приглашением MS DOS. Перемещайтесь по каталогам, используя команду cd <имя каталога>. Для того чтобы запустить любые откомпилированные программы, в том числе те, тексты которых имеются в книге, убедитесь, что вы находитесь в том же каталоге, что и нужный файл .EXE, и наберите имя программы. Более подробную информацию можно получить, воспользовавшись Справочной системой Windows.

Предварительно скомпилированные заголовочные файлы

Процесс компиляции можно существенно ускорить, если использовать предварительно откомпилированные заголовочные файлы. Для этого нужно выбрать пункт Options (Параметры) из меню Project (Проект), перейти к закладке Compiler (Компилятор) и установить флажок Use Precompiled Headers (Использовать прекомпилированные заголовоные файлы). В коротких программах больше всего времени тратится на компиляцию служебных заголовочных файлов C++, таких, как iostream. Включение указанного параметра позволяет скомпилировать эти файлы только один раз.

Закрытие и открытие проектов

Когда вы заканчиваете работу над проектом, его необходимо закрыть, выбрав пункт Close All (Закрыть все) из меню File (Файл). Чтобы открыть уже имеющий-

ся проект, нужно из того же меню выбрать пункт Open Project (Открыть проект), перейти в соответствующую директорию и дважды щелкнуть на имени файла с расширением .BPR.

Добавление заголовочного файла к проекту

Серьезные программы, написанные на C++, могут работать с одним или даже несколькими пользовательскими заголовочными файлами (и это — в дополнение к множеству библиотечных файлов, таких, как IOSTREAM и CONIO.H). Покажем, как добавить их к проекту.

Создание нового заголовочного файла

Выберите пункт New... (Новый...) из меню File (Файл), убедитесь, что вы находитесь на закладке, в которой отображаются новые элементы, щелкните дважды на значке Text (Текст). Появится окно редактора кода с файлом FILE1.TXT. Наберите текст и сохраните файл с помощью пункта Save as... (Сохранить как...) из меню File (Файл). Придумайте имя для файла, но не забудьте оставить расширение .H. Этот файл нужно сохранить в той же директории, где находятся ваши исходные (.CPP) файлы. Имя сохраненного файла появится в виде одной из закладок окна редактора. Теперь для редактирования разных файлов можно просто переключаться между закладками.

Редактирование заголовочного файла

Чтобы открыть уже существующий заголовочный файл, выберите Open (Открыть) из меню File (Файл), укажите в поле Files of Type (Типы файлов) значение *.* (Все файлы). Теперь можно выбрать из списка нужный заголовочный файл.

При написании выражения #include для заголовочного файла в исходном не забудьте убедиться в том, что имя файла заключено в кавычки:

```
#include "myHeader.h"
```

Эти кавычки укажут компилятору, что он должен искать заголовочный файл в том же каталоге, где находятся исходные файлы.

Определение местонахождения заголовочного файла

Если уж вы добавляете файл .H, то нужно сообщить компилятору о его местонахождении. Если он находится в директории с другими вашими файлами, относящимися к данному проекту, то ничего делать не нужно.

В противном случае нужно указать его расположение. Например, это необходимо делать в том случае, если вы используете консольную графику в своей

программе. Нужно указать расположение файла BORLACON.H, если вам лень скопировать его в каталог с остальными файлами. Перейдите к пункту Options (Параметры) меню Project (Проект) и выберите закладку Directories/Conditionals (Каталоги/Импликации). В секции Directories (Каталоги) щелкните на кнопочке с тремя точками, расположенной справа от списка включаемых путей Include Path. Появится диалоговое окно Directories (Каталоги).

В самом низу этого окна наберите полный путь к каталогу, содержащему заголовочный файл. Щелкните на кнопке Add (Добавить), чтобы занести этот каталог в список включаемых путей. Понажимайте кнопки OK, чтобы закрылись все диалоги.

Не пытайтесь добавлять заголовочные файлы с помощью функции Add to Project (Добавить в проект) из меню Project (Проект).

Проекты с несколькими исходными файлами

Реальные приложения и даже некоторые примеры из книги требуют подключения нескольких исходных файлов. В C++ Builder исходные файлы называются *компонентами* (units), это название характерно именно для данной среды программирования. В большинстве других сред файлы все-таки называются файлами, в крайнем случае, *модулями*.

Создание дополнительных исходных файлов

Дополнительные .CPP-файлы можно создавать так же, как заголовочные файлы. Для этого нужно выбрать File ▸ New (Файл ▸ Новый) и дважды щелкнуть на значке Text (Текст) в диалоговом окне New (Новые). Наберите исходный код программы и сохраните файл. При сохранении не забудьте указать правильный тип: это должен быть C++ Builder Unit (компонент C++ Builder). Расширение .CPP подставится автоматически после имени файла. Если почему-то этого не случилось и вы просто наберете имя файла и даже правильное расширение, то этот файл не распознается как C++ Builder unit.

Добавление существующих исходных файлов

У вас может уже быть созданный исходный файл, например BORLACON.CPP, необходимый для работы с консольной графикой. Чтобы добавить исходный файл в проект, выберите пункт Add To Project (Добавить в проект) из меню Project (Проект), перейдите в требуемую директорию и выберите файл, который хотите добавить. Затем нажмите кнопку Open (Открыть). Теперь этот файл в C++ Builder будет считаться частью проекта.

Исходные файлы указаны в закладках окна редактирования, так что можно быстро переключаться с одного файла на другой. Некоторые из них можно закрывать, чтобы они не занимали место на экране, если они не нужны.

Менеджер проектов

Чтобы посмотреть, какие файлы являются составными частями проекта, нужно выбрать Project Manager (Менеджер проектов) из меню View (Вид). Вы увидите диаграмму, на которой показаны связи между файлами, примерно как в Проводнике Windows. Нажав на плюс рядом со значком проекта, можно увидеть все файлы, входящие в проект. Файл, который вы только что добавили в проект, должен быть среди них.

Если нажать правой кнопкой мыши на Project Manager (Менеджере проектов), появится контекстное меню, из которого можно выбрать следующие действия: Open (Открыть), Close (Закрыть), Save as... (Сохранить как...) и Compile (Компилировать). Это удобный способ работы с отдельными исходными файлами.

В многофайловых программах можно компилировать исходные файлы независимо друг от друга. Для этого нужно выбрать пункт Compile Unit (Компилировать Компонент) из меню Project (Проект). При этом перекомпилируются только те файлы, которые были изменены со времени предыдущей сборки программы.

Программы с консольной графикой

Опишем, как осуществлять компоновку программ, в которых используется консольная графика. Это относится, в частности, к программам CIRCSTRC из главы 5 «Функции» и CIRCLES из главы 6 «Объекты и классы». Итак, по порядку.

- Создайте новый проект, как было описано выше. В качестве имени проекта лучше всего использовать имя программы. Поменяйте только расширение на .BPR.
- Затем в исходном файле поменяйте строку #include <msoftcon.h> на #include <borlacon.h>.
- Скопируйте (именно скопируйте, а не переносите) файлы BORLACON.CPP и BORLACON.H в файл проекта (или же сообщите компилятору, где искать заголовочный файл).
- Добавьте исходный файл BORLACON.CPP в свой проект, следуя инструкциям, описанным чуть выше в разделе «Добавление существующих исходных файлов».
- Чтобы задержать окно результатов программы на экране, вставьте строку getch(); непосредственно перед последним оператором return в конце секции main().
- Для поддержки getch() нужно включить в программу еще один заголовочный файл. Для этого напишите в начале программы строку #include <conio.h>.

Теперь можно компилировать, связывать и запускать программы с использованием консольной графики точно так же, как любые другие.

Отладка

В главе 3 «Циклы и ветвления» мы использовали отладчик в основном с целью более полного ознакомления с процессом работы циклов. При использовании C++ Builder те же самые шаги могут помочь вам найти ошибку в программе и устранить ее. Мы будем рассматривать лишь однофайловые программы, но следует отметить, что отладка многофайловых проектов ничем принципиально не отличается.

Для начала просто создайте и откомпилируйте какую-либо программу. Избавьтесь от всех ошибок компилятора и компоновщика. Убедитесь, что листинг программы выводится в окошке редактирования.

Пошаговый прогон

Чтобы запустить отладчик, нажмите клавишу F8. Вся программа будет перекомпилирована, а первая ее строка (обычно описатель main()) — подсвечена. Последовательные нажатия F8 приведут к продвижению по тексту программы от одного оператора к другому. Если вы прогоняете цикл, то увидите, как стрелка доходит до последнего выражения, а потом переходит вновь на начало цикла.

Просмотр переменных

Существует возможность просмотра значений переменных в процессе пошаговой трассировки. Выберите пункт Add watch (Добавить переменную) из меню Run (Пуск). Появится диалоговое окно Watch Properties (Параметры просмотра). Наберите имя переменной в поле Выражение, затем выберите ее тип и нажмите OK. Появится окно Watch List (Список наблюдения). Выполняя указанные шаги несколько раз, можно добавить в этот список любое необходимое число переменных. Если вы нормально разместите на экране окно просмотра и окно редактирования, то сможете наблюдать одновременное изменение значений переменных в зависимости от выполняемой в данный момент команды. Если переменная пока недоступна (например, еще не определена в программе), окно просмотра выдаст сообщение об ошибке вместо значения рядом с именем переменной.

В случае с программой CUBELIST механизм наблюдения переменных не распознает переменную cube, так как она определяется внутри цикла. Перепишите программу таким образом, чтобы она определялась вне цикла, и сразу увидите результат: в окне наблюдения она отображается, как следует.

Пошаговая трассировка функций

Если в вашей программе используются функции, то имейте в виду, что есть замечательная возможность осуществить пошаговую трассировку каждого выражения функции. Это делается с помощью клавиши F7. В отличие от режима трассировки по F8, который рассматривает каждую встреченную функцию как

одно выражение, такой режим позволяет «войти» внутрь функции и выявить возможные ошибки в ней. Если вы заставите по F7 проходить библиотечные функции, например cout <<, то будете любоваться исходным кодом библиотеки. Это процесс долгий, поэтому советуем пользоваться этой возможностью, только если вам действительно важно знать, что происходит внутри рутинных функций. Впрочем, можно по мере необходимости чередовать нажатия F7 и F8, в зависимости от необходимости выявления ошибок во внутренних функциях.

Точки останова

Точки останова позволяют временно прерывать работу программы в произвольных местах. Когда их нужно использовать? Мы уже показали, как можно заставить отладчик прогонять программу только до курсора. Но бывают случаи, когда необходимо иметь несколько таких точек. Например, можно остановить программу в конце условия if, а также после соответствующего else. Точки останова решают эту задачу, поскольку их можно ставить в неограниченном количестве. У них есть еще и расширенные функции, которые мы здесь рассматривать не будем.

Рассмотрим, как вставить точки останова в листинги. Во-первых, взгляните на листинг программы в окне редактирования. Напротив каждого выполняемого оператора вы увидите точечку слева. Просто щелкните мышью на той точечке, напротив которой вы собираетесь вставить точку останова. Точечка заменится на красный кружочек, а строка кода будет подсвечена. Теперь при любом варианте запуска программы в среде программа будет останавливаться именно здесь. Можно на этом временном срезе проверить значения переменных, произвести пошаговую трассировку кода и т. д.

Чтобы удалить точку останова, щелкните на красном кружочке. Он исчезнет.

Есть еще множество других функций отладчика; мы показали лишь основные, чтобы вам было с чего начать работу.

Приложение Д Упрощенный вариант консольной графики

- Использование подпрограмм библиотеки консольной графики
- Функции библиотеки консольной графики
- Реализация функций консольной графики
- Листинги исходных кодов

Было бы здорово немного оживить примеры из книги какой-нибудь графикой, поэтому мы включили несколько примеров, использующих графические функции. Стандартный C++ не включает в себя спецификации графических подпрограмм, но графика в этом языке не является чем-то табуированным. A Windows, как известно, поддерживает самые разнообразные типы графики.

Microsoft Visual C++ и Borland C++ используют разные библиотечные функции для реализации графики в программах, и, конечно же, ни одна из них полностью не сможет удовлетворить всем нашим требованиям. Чтобы не мучиться с различиями в использовании библиотек разными средами программирования и расширить графические возможности и удобство работы, мы предлагаем использовать свой набор функций, который мы назвали Упрощенным вариантом консольной графики. Эти функции подходят и для среды Borland, и для среды Microsoft. Для реализации этой возможности используются два различных набора файлов: MSOFTCON.H и MSOFTCON.CPP для Microsoft Visual C++, и BORLACON.H и BORLACON.CPP для Borland C++ Builder (вполне может быть, что некоторые функции, реализованные для среды Microsoft, будут обработаны и другими компиляторами).

Все файлы, входящие в набор библиотеки упрощенного варианта консольной графики, можно скачать с нашего сайта. Если вы скачивали какие-нибудь исходные коды примеров, значит, у вас, скорее всего, уже есть эта библиотека. Если это не так, смотрите инструкции по скачиванию файлов во введении к книге. Листинги всех этих файлов входят в состав данного приложения.

Наши графические подпрограммы используют консольную графику. Консольный режим — это текстовый режим, имеющий обычно размеры 80 столбцов на 25 строк. Большинство примеров (не графических) в нашей книге используют вывод текста в консольном режиме. Программа, являющаяся консольной, мо-

жет выводить результаты в собственном окне в Windows или быть вообще отдельно стоящей программой MS DOS.

В консольном режиме всяческие кружочки, прямоугольники и т. п. создаются из символов (например, из буковок «X» или квадратиков), а не из пикселов. Результаты, конечно, не очень-то радуют глаз, особенно привыкший к современной компьютерной графике высокого качества, но для демонстрационных программ такой вид вполне подходит.

Использование подпрограмм библиотеки консольной графики

Чтобы создать программу, использующую графические функции, нужно проделать несколько дополнительных шагов по сравнению с обычной процедурой сборки. Они таковы:

- включить соответствующий заголовочный файл (MSOFTCON.H или BORLACON.H) в исходный файл (.CPP);
- добавить в проект соответствующий исходный файл (MSOFTCON.CPP или BORLACON.CPP), чтобы он при компоновке оказался связанным с программой;
- Убедиться в том, что компилятор может найти заголовочный и исходный файлы.

Заголовочный файл содержит объявления функций Упрощенного варианта консольной графики. Исходные же файлы содержат их определения. Вам необходимо скомпилировать исходный файл и связать получившийся в результате .OBJ-файл со всеми остальными частями программы. Собственно говоря, это происходит автоматически при добавлении исходного файла в проект.

Чтобы узнать, как подключить файл к проекту, читайте приложение В «Microsoft Visual C++» или приложение Г «Borland C++ Builder». Затем примените полученные знания для подключения к своей программе исходных файлов библиотеки консольной графики.

Вам может потребоваться прописать путь к нужному заголовочному файлу в параметре Директории своей среды программирования, чтобы компилятор был в состоянии найти его. Вся необходимая информация о том, как это делается, находится в указанных приложениях.

Вот и все, что вам нужно проделать, чтобы просто иметь возможность запустить графические примеры из книги. Если же вы хотите использовать графические функции нашей самодельной библиотеки в своих программах, читайте дальше.

Функции библиотеки консольной графики

Функции нашей библиотеки предполагают наличие консольного режима экрана размером 80 столбцов на 25 строк. Верхний левый угол имеет координаты (1,1), а правый нижний — (80,25).

Эти функции были созданы специально для демонстрационных программ из этой книги и не отличаются особой сложностью и гениальностью. Если вы намерены использовать их в своих программах, необходимо внимательно следить, чтобы все рисуемые фигуры помещались на экране 80×25. Если вы будете использовать некорректные координаты, поведение функций станет непредсказуемым. В табл. Д.1 приводится список всех функций нашей скромной библиотеки.

Таблица Д.1. Функции библиотеки упрощенного варианта консольной графики

Имя функции	Назначение
init_graphics()	Инициализация графической системы
set_color()	Установка цвета переднего и заднего плана
set_cursor_pos()	Установка курсора на указанный столбец и строку
clear_screen()	Очистка экрана
wait(n)	Приостановка программы на n миллисекунд
clear_line()	Очистка строки
draw_rectangle()	Задание прямоугольника по координатам: верхний левый, правый нижний углы
draw_circle()	Задание окружности по координатам центра и радиусу
draw_line()	Задание отрезка по начальной и конечной точкам
draw_pyramid()	Задание координаты вершины и высоты
set_fill_style()	Задание символа заполнения

Перед началом использования графических функций всегда нужно вызывать функцию инициализации графической системы init_graphics(). Она задает символ заполнения, а в Microsoft-версии также инициализирует некоторые другие важные элементы графической системы.

У функции set_color() может быть один или два аргумента. Первый задает цвет переднего плана, а второй (необязательный) задает цвет заднего плана (фона символа). В большинстве случаев цвет заднего фона оставляют черным.

```
set_color(cRED);          //передний план - красный
set_color(cWHITE, cBLUE); //белые буквы на синем фоне
```

Приведем список цветовых констант для set_color():

```
cBLACK
cDARK_BLUE
cDARK_GREEN
cDARK_CYAN
cDARK_RED
cDARK_MAGENTA
cBROWN
cLIGHT_GRAY
cDARK_GRAY
cBLUE
cGREEN
cCYAN
cRED
cMAGENTA
cYELLOW
cWHITE
```

Функции, начинающиеся с draw_, рисуют разные геометрические фигуры или линии с использованием специального символа, называемого *символом заполнения*. По умолчанию символом заполнения является маленький сплошной прямоугольник, но с помощью функции set_fill_style() можно назначить другой символ. Это может быть один из символов «О», «Х» или один из трех затененных символов. Список констант символов заполнения:

```
SOLID_FILL
X_FILL
O_FILL
LIGHT_FILL
MEDIUM_FILL
DARK_FILL
```

Аргументом функции wait() является время задержки в миллисекундах.

```
wait(3000):                 //задержка на 3 секунды
```

Все прочие функции, видимо, понятны без дополнительных объяснений. Они используются в наших примерах, так что можно изучать их прямо на практике.

Реализация функций консольной графики

Все эти функции не имеют, строго говоря, отношения ни к какому объектно-ориентированному программированию, поэтому написать их можно было бы как на языке С, так и на С++. Поэтому никакого смысла изучать их в подробностях мы не видим, разве что сами читатели вдруг заинтересуются таким поспешным и смешным подходом к реализации графических операций, состоящим из рисования черточек и кружочков.

Идея этой маленькой библиотеки состояла в том, чтобы сделать графические функции минимальными средствами. Если вас это очень заинтересует, можете изучить исходные файлы, включенные в конец данного приложения.

Компиляторы Microsoft

Компиляторы Microsoft больше не содержат встроенных графических функций, как это было еще несколько лет назад. Дело в том, что операционная система Windows сама по себе является чисто графической и предоставляет огромный собственный набор инструментальных средств для осуществления простых консольных графических операций, таких, как позиционирование курсора, изменение цвета текста и т. п. Для работы с компиляторами Microsoft функции нашей библиотеки консольной графики обращаются именно к этим встроенным функциям Windows. (Благодарим Andre LaMothe за идею такого решения. Его прекрасная книга имеется в списке рекомендуемой литературы, см. приложение З «Библиография».)

Для внедрения в свою программу функций консольной графики необходимо создать проект типа «Win32 Console Application», как описано в приложении В.

Консольные функции Windows не будут работать до тех пор, пока вы не инициализируете графическую систему нужным образом. Для этого необходимо вызвать функцию init_graphics().

Компиляторы Borland

Компиляторы фирмы Borland все еще включают в себя встроенные графические функции как для консольных приложений, так и для пиксельной графики. Если вы пользуетесь файлом BORLACON.CPP, то функции библиотеки упрощенного варианта консольной графики транслируются в соответствующие функции компилятора Borland.

Может возникнуть вопрос, почему нельзя использовать компилятор фирмы Borland и иметь доступ к консольным функциям, встроенным в Windows. Проблема состоит в том, что для создания консольной программы в Borland C++ нужен или EasyWin, или DOS. И одна, и другая являются 16-битными системами, в то время как консольные функции Windows 32-битные и не могут использоваться с консольным режимом в понимании Borland.

При работе с Borland C++ потоковый ввод/вывод с помощью iostream (cout <<) не работает с разными цветами. Из-за этого некоторые демонстрационные программы, например HORSE.CPP, используют функции консольного режима: cputs(), putch(), располагающиеся в заголовочном файле CONIO.H.

Листинги исходных кодов

Представим листинги четырех файлов, из которых состоит библиотека упрощенной консольной графики: MSOFTCON.H и MSOFTCON.CPP для Microsoft; BORLACON.H и BORLACON.CPP для C++ Builder. В обычных ситуациях вам не придется вникать в содержимое этих файлов, они приведены здесь скорее для того, чтобы было на что сослаться.

Листинг Д.1. Листинг MSOFTCON.H[1]

```
//msoftcon.h
//Объявления функций консольной графики от Lafore
//используются консольные функции Windows

#ifndef _INC_WCONSOLE      //этот файл не должен включаться
#define _INC_WCONSOLE      //дважды в тот же исходный файл

#include <windows.h>       //для консольных функций Windows
#include <conio.h>         //для kbhit(), getche()
#include <math.h>          //для sin, cos

enum fstyle { SOLID_FILL, X_FILL,      O_FILL,
              LIGHT_FILL, MEDIUM_FILL, DARK_FILL };

enum color {
```

[1] Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download.

```
   cBLACK=0.     cDARK_BLUE=1.  cDARK_GREEN=2. DARK_CYAN=3.
   cDARK_RED=4.  cDARK_MAGENTA=5. cBROWN=6.  cLIGHT_GRAY=7.
   cDARK_GRAY=8. cBLUE=9.          cGREEN=10.     cCYAN=11.
   cRED=12.      cMAGENTA=13.      cYELLOW=14.    cWHITE=15 };
//----------------------------------------------------------
void init_graphics();
void set_color(color fg. color bg = cBLACK);
void set_cursor_pos(int x. int y);
void clear_screen();
void wait(int milliseconds);
void clear_line();
void draw_rectangle(int left. int top. int right. int bottom);
void draw_circle(int x. int y. int rad);
void draw_line(int x1. int yl. int x2. int y2);
void draw_pyramid(int xl. int yl. int height);
void set_fill_style(fstyle);
#endif /* _INC_WCONSOLE */
```

Листинг Д.2. Листинг MSOFTCON.CPP

```
//msoftcon.cpp
//подпрограммы доступа к консольным функциям Windows

//компилятор должен знать. где искать этот файл
//в MCV++. /Tools/Options/Directories/Include/type путь

#include "msoftcon.h"
HANDLE hConsole;          //поддержка консольного режима
char fill_char;           //символ заполнения
//----------------------------------------------------------
void init_graphics()
   {
   COORD console_size = {80. 25};
   //открыть канал ввода/вывода на консоль
   hConsole = CreateFile("CONOUT$". GENERIC_WRITE | GENERIC_READ.
                  FILE_SHARE_READ | FILE_SHARE_WRITE.
                  0L. OPEN_EXISTING. FILE_ATTRIBUTE_NORMAL. 0L);
   //установить размер экрана 80x25
   SetConsoleScreenBufferSize(hConsole. console_size);
   //текст белым по черному
   SetConsoleTextAttribute( hConsole. (WORD)((0 << 4) | 15) );

   fill_char = '\xDB';  //заполнение по умолчанию
   clear_screen();
   }
//----------------------------------------------------------
void set_color(color foreground. color background)
   {
   SetConsoleTextAttribute( hConsole.
                      (WORD)((background << 4) | foreground) );
   }  //конец setcolor()

/* 0  Черный          8  Темно-серый
   1  Темно-синий     9  Синий
   2  Темно-зеленый   10 Зеленый
   3  Темно-голубой   11 Алый
   4  Темно-красный   12 красный
   5  Темно-алый      13 алый
```

продолжение ⇨

Листинг Д.2 (*продолжение*)

```
   6  Коричневый      14 Желтый
   7  Светло-серый    15 Белый
*/
//--------------------------------------------------------------
void set_cursor_pos(int x, int y)
   {
   COORD cursor_pos;             //Начало в верхнем левом
   cursor_pos.X = x - 1;         //Windows начинает с (0, 0)
   cursor_pos.Y = y - 1;         //мы начнем с (1, 1)
   SetConsoleCursorPosition(hConsole, cursor_pos);
   }
//--------------------------------------------------------------
void clear_screen()
   {
   set_cursor_pos(1, 25);
   for(int j=0; j<25; j++)
      putch('\n');
   set_cursor_pos(1, 1);
   }
//--------------------------------------------------------------
void wait(int milliseconds)
   {
   Sleep(milliseconds);
   }
//--------------------------------------------------------------
void clear_line()                 //очистка до конца строки
   {                              //80 пробелов
   //.....1234567890123456789012345678901234567890
   //.....0.........1.........2.........3.........4
   cputs("                                        ");
   cputs("                                        ");
   }
//--------------------------------------------------------------
void draw_rectangle(int left, int top, int right, int bottom)
   {
   char temp[80];
   int width = right - left + 1;

   for(int j=0; j<width; j++)      //строка квадратов
      temp[j] = fill_char;
   temp[j] = 0;                    //null

   for(int y=top; y<=bottom; y++)  //строковый стек
      {
      set_cursor_pos(left, y);
      cputs(temp);
      }
   }
//--------------------------------------------------------------
void draw_circle(int xC, int yC, int radius)
   {
   double theta, increment, xF, pi=3.14159;
   int x, xN, yN;

   increment = 0.8 / static_cast<double>(radius);
   for(theta=0;theta<=pi/2;theta+=increment)//1/4
                                               //окружности
```

```
      {
      xF = radius * cos(theta);
      xN = static_cast<int>(xF * 2 / 1); //пикселы не
                                          //квадратные :-(
      yN = static_cast<int>(radius * sin(theta) + 0.5);
      x = xC-xN;
      while(x <= xC+xN) //заполнить две горизонтальные линии
         {              //по одной на каждую ? окружности
         set_cursor_pos(x, yC-yN); putch(fill_char); //верх
         set_cursor_pos(x++,yC+yN); putch(fill_char); //низ
         }
      }  //конец for
   }
//-----------------------------------------------------------
void draw_line(int x1, int y1, int x2, int y2)
   {

   int w, z, t, w1, w2, z1, z2;
   double xDelta=x1-x2, yDelta=y1-y2, slope;
   bool isMoreHoriz;

   if( fabs(xDelta) > fabs(yDelta) ) //еще горизонтальная
      {
      isMoreHoriz = true;
      slope = yDelta / xDelta;
      w1=x1; z1=y1; w2=x2, z2=y2;    //w=x, z=y
      }
   else                              //еще вертикальная
      {
      isMoreHoriz = false;
      slope = xDelta / yDelta;
      w1=y1; z1=x1; w2=y2, z2=x2;    //w=y, z=x
      }

   if(w1 > w2)                       //если за w
      {
      t=w1; w1=w2; w2=t;             //   заменить (w1,z1)
      t=z1; z1=z2; z2=t;             //   на (w2,z2)
      }
   for(w=w1; w<=w2; w++)
      {
      z = static_cast<int>(z1 + slope * (w-w1));
      if( !(w==80 && z==25) )  //убрать прокрутку на 80.25
         {
         if(isMoreHoriz)
            set_cursor_pos(w, z);
         else
            set_cursor_pos(z, w);
         putch(fill_char);
         }
      }
   }
//-----------------------------------------------------------
void draw_pyramid(int x1, int y1, int height)
   {
   int x, y;
   for(y=y1; y<y1+height; y++)
```

продолжение

Листинг Д.2 (*продолжение*)

```
      {
      int incr = y - y1;
      for(x=x1-incr; x<=x1+incr; x++)
         {
         set_cursor_pos(x, y);
         putch(fill_char);
         }
      }
   }
//-----------------------------------------------------------
void set_fill_style(fstyle fs)
   {
   switch(fs)
      {
      case SOLID_FILL:  fill_char = '\xDB'; break;
      case DARK_FILL:   fill_char = '\xB0'; break;
      case MEDIUM_FILL: fill_char = '\xB1'; break;
      case LIGHT_FILL:  fill_char = '\xB2'; break;
      case X_FILL:      fill_char = 'X';    break;
      case O_FILL:      fill_char = 'O';    break;
      }
   }
//-----------------------------------------------------------
```

Листинг Д.3. Листинг BORLACON.H

```
//borlacon.h
//объявления функций упрощенного варианта консольной графики
//используются консольные функции Borland
#ifndef _INC_WCONSOLE  //не включать этот файл дважды
#define _INC_WCONSOLE  //в один исходный файл

#include <windows.h>   //для Sleep()
#include <conio.h>     //для kbhit(), getche()
#include <math.h>      //для sin, cos

enum fstyle { SOLID_FILL, X_FILL, O_FILL,
              LIGHT_FILL, MEDIUM_FILL, DARK_FILL };

enum color {
   cBLACK=0,     cDARK_BLUE=1, cDARK_GREEN=2, DARK_CYAN=3,
   cDARK_RED=4, cDARK_MAGENTA=5,  cBROWN=6, cLIGHT_GRAY=7,
   cDARK_GRAY=8, cBLUE=9,         cGREEN=10,     cCYAN=11,
   cRED=12,      cMAGENTA=13,     cYELLOW=14,    cWHITE=15 };
//-----------------------------------------------------------
void init_graphics();
void set_color(color fg, color bg = cBLACK);
void set_cursor_pos(int x, int y);
void clear_screen();
void wait(int milliseconds);
void clear_line();
void draw_rectangle(int left, int top, int right, int bottom);
void draw_circle(int x, int y, int rad);
void draw_line(int x1, int y1, int x2, int y2);
void draw_pyramid(int x1, int y1, int height);
void set_fill_style(fstyle);
#endif // _INC_WCONSOLE
```

Листинг Д.4. Листинг BORLACON.CPP

```
//borlacon.cpp
//подпрограммы рисования для консольных функций Borland
#include "borlaCon.h"

char fill_char;              //символ заполнения
//----------------------------------------------------------
void init_graphics()
   {
   textcolor(WHITE);         //текст белый по черному
   textbackground(BLACK);
   fill_char = '\xDB';       // заполнение по умолчанию
   clrscr();
   }
//----------------------------------------------------------
void set_color(color foreground, color background)
   {
   textcolor( static_cast<int>(foreground) );
   textbackground( static_cast<int>(background) );
   }
//----------------------------------------------------------
void set_cursor_pos(int x, int y)
   {
   gotoxy(x, y);
   }
//----------------------------------------------------------
void clear_screen()
   {
   clrscr();
   }
//----------------------------------------------------------
void wait(int milliseconds)
   {
   Sleep(milliseconds);
   }
//----------------------------------------------------------
void clear_line()                 // очистка до конца строки
   {                              // 80 пробелов
   //.....1234567890123456789012345678901234567890
   //.....0.........1.........2.........3.........4
   cputs("                                        ");
   cputs("                                        ");
   } //конец clreol()
//----------------------------------------------------------
void draw_rectangle(int left, int top, int right, int bottom)
   {
   int j;
   char temp[80];
   int width = right - left + 1;

   for(j=0; j<width; j++)         //строка квадратиков
      temp[j] = fill_char;
   temp[j] = 0;                   //null

   for(int y=top; y<=bottom; y++) //строковый стек
      {
```

продолжение ➩

Листинг Д.4 (*продолжение*)

```
      set_cursor_pos(left, y);
      cputs(temp);
      }
   }  //конец rectangle
//--------------------------------------------------------------
void draw_circle(int xC, int yC, int radius)
   {
   double theta, increment, xF, pi=3.14159;
   int x, xN, yN;

   increment = 0.8 / static_cast<double>(radius);
   for(theta=0; theta<=pi/2; theta+=increment)  //четверть
                                                //круга
      {
      xF = radius * cos(theta);
      xN = static_cast<int>(xF * 2 / 1); //пикселы не
                                         //квадратные
      yN = static_cast<int>(radius * sin(theta) + 0.5);
      x = xC-xN;
      while(x <= xC+xN)    //заполнить две гориз. линии
         {                 //по одной на каждую полуокружность
         set_cursor_pos(x,  yC-yN); putch(fill_char);//верх
         set_cursor_pos(x++, yC+yN); putch(fill_char);//низ
         }
      }  //конец for
   }  //конец circle()
//--------------------------------------------------------------
void draw_line(int x1, int y1, int x2, int y2)
   {

   int w, z, t, w1, w2, z1, z2;
   double xDelta=x1-x2, yDelta=y1-y2, slope;
   bool isMoreHoriz;

   if( fabs(xDelta) > fabs(yDelta) ) //еще горизонтальная
      {
      isMoreHoriz = true;
      slope = yDelta / xDelta;
      w1=x1; z1=y1; w2=x2, z2=y2;    //w=x, z=y
      }
   else                              //еще вертикальная
      {
      isMoreHoriz = false;
      slope = xDelta / yDelta;
      w1=y1; z1=x1; w2=y2, z2=x2;    //w=y, z=x
      }

   if(w1 > w2)                       //если за w
      {
      t=w1; w1=w2; w2=t;             //    поменять (w1,z1)
      t=z1; z1=z2; z2=t;             //    на (w2,z2)
      }
   for(w=w1; w<=w2; w++)
      {
      z = static_cast<int>(z1 + slope * (w-w1));
      if( !(w==80 && z==25)) //запретить прокрутку на 80,25
         {
```

```
        if(isMoreHoriz)
           set_cursor_pos(w, z);
        else
           set_cursor_pos(z, w);
        putch(fill_char);
        }
     }
  }
//-----------------------------------------------------
void draw_pyramid(int x1, int y1, int height)
  {
  int x, y;
  for(y=y1; y<y1+height; y++)
     {
     int incr = y - y1;
     for(x=x1-incr; x<=x1+incr; x++)
        {
        set_cursor_pos(x, y);
        putch(fill_char);
        }
     }
  }
//-----------------------------------------------------
void set_fill_style(fstyle fs)
  {
  switch(fs)
     {
     case SOLID_FILL:  fill_char = '\xDB'; break;
     case DARK_FILL:   fill_char = '\xB0'; break;
     case MEDIUM_FILL: fill_char = '\xB1'; break;
     case LIGHT_FILL:  fill_char = '\xB2'; break;
     case X_FILL:      fill_char = 'X';    break;
     case O_FILL:      fill_char = 'O';    break;
     }
  }
//-----------------------------------------------------
```

Приложение Е
Алгоритмы и методы STL

- Алгоритмы
- Методы
- Итераторы

В этом приложении содержатся таблицы алгоритмов и методов контейнеров Стандартной библиотеки шаблонов (STL). Представленная в них информация базируется на описании STL (1995), авторами которого являются Александр Степанов и Минг Ли, но мы, разумеется, сильно сократили и переработали это описание, позволив себе некоторые вольности в формулировках в интересах более компактного изложения материала.

Алгоритмы

В табл. Е.1 показаны алгоритмы STL. Приводятся краткие описания того, что они делают. Описания не претендуют на роль математически точных определений. За более подробной информацией, включая типы данных, используемых в качестве аргументов, и типы возвращаемых значений, обращайтесь к книгам, указанным в приложении З «Библиография».

В первой колонке таблицы приводятся имена функций, во второй содержатся краткие описания алгоритмов, а в последней колонке — список аргументов. Тип данных, возвращаемых алгоритмами, явно не указывается. Тем не менее о нем можно догадаться по описанию, и он либо очевиден, либо не столь принципиален.

В колонке аргументов вы встретите такие значения: `first`, `last`, `first1`, `last1`, `first2`, `last2`, `first3` и `middle`. Это итераторы, указывающие на определенные позиции в контейнерах. С помощью идентификаторов `first1` и `last1` ограничивается диапазон 1, соответственно, с помощью `first2` и `last2` — диапазон 2. Аргументы `function`, `predicate`, `op` и `comp` — это функциональные объекты. Аргументы `value`, `old`, `new` и `init` — значения объектов, хранящихся в контейнерах. Эти значения сортируются или сравниваются с помощью операций <, == или функционального объекта `comp`.

В колонке «Назначение» перемещаемые итераторы обозначаются идентификаторами iter1, iter2 и iter. Когда используются одновременно итераторы iter1 и iter2, предполагается, что они оба перемещаются, каждый по своему контейнеру (или по двум диапазонам одного и того же контейнера).

Таблица Е.1. Алгоритмы STL

Название	Назначение	Аргументы
Не изменяющие последовательные операции		
for_each	Применяет function ко всем объектам	first, last, function
find	Возвращает итератор, указывающий на первый объект, равный значению value	first, last, value
find_if	Возвращает итератор, указывающий на первый объект, для которого predicate принимает значение «истина»	first, last, predicate
adjacent_find	Возвращает итератор, указывающий на первую пару равных объектов	first, last
adjacent_find	Возвращает итератор, указывающий на первую пару объектов, удовлетворяющих значению predicate	first, last, predicate
count	Прибавляет к n число объектов, равных значению value	first, last, value, n
count_if	Прибавляет к n число объектов, удовлетворяющих значению predicate	first, last, predicate, n
mismatch	Возвращает первую несовпадающую пару соответствующих объектов, расположенных в разных диапазонах позиций контейнера	first1, last1, first2
mismatch	Возвращает первую пару соответствующих объектов, расположенных в разных диапазонах позиций контейнера, не удовлетворяющих predicate	first1, last1, first2, predicate
equal	Возвращает значение «истина», если все соответствующие пары объектов из двух различных диапазонов равны	first1, last1, first2
equal	Возвращает значение «истина», если все соответствующие пары объектов из двух различных диапазонов удовлетворяют predicate	first1, last1, first2, predicate
search	Проверяет, содержится ли второй диапазон внутри первого. Возвращает начало совпадения или last1, если нет совпадения	first1, last1, first2, last2
search	Проверяет, содержится ли второй диапазон внутри первого, равенство определяется по predicate. Возвращает начало совпадения или last1, если нет совпадения	first1, last1 , first2, last2, predicate
Изменяющие последовательные операции		
copy	Копирует объекты из первого диапазона во второй	first1, last1, first2
copy_backward	Копирует объекты из первого диапазона во второй, располагая их в обратном порядке, от last2 к first2	first1, last1, first2
swap	Заменяет один объект другим	a, b
iter_swap	Обменивает объекты, на которые указывают два итератора	iter1, iter2

продолжение

Таблица Е.1 (*продолжение*)

Название	Назначение	Аргументы
swap_ranges	Обменивает соответствующие объекты в двух диапазонах	first1, last1, first2
transform	Превращает объекты из диапазона 1 в новые объекты диапазона 2, применяя operator	first1, last1, first2, operator
transform	Комбинирует объекты из диапазонов 1 и 2, создавая новые объекты в диапазоне 3, применяя operator	first1, last1, first2, first3, operator
replace	Заменяет все объекты, равные old, объектами, равными new	first, last, old, new
replace_if	Заменяет все объекты, удовлетворяющие predicate, объектами, равными new	first, last, predicate, new
replace_copy	Производит копирование из диапазона 1 в диапазон 2, заменяя все объекты, равные old на объекты, равные new	first1, last1, first2, old, new
replace_copy_if	Производит копирование из диапазона 1 в диапазон 2, заменяя все объекты, удовлетворяющие predicate, на объекты, равные new	first1, last1, first2, predicate, new
fill	Присваивает значение value всем объектам из диапазона	first, last, value
fill_n	Присваивает значение value всем объектам из диапазона от first до first+n	first, n, value
generate	Заполняет диапазон значениями, получаемыми с помощью последовательных вызовов функции gen	first, last, gen
generate_n	Заполняет диапазон от first до first+n значениями, получаемыми с помощью последовательных вызовов функции gen	first, n, gen
remove	Удаляет из диапазона все объекты, равные value	first, last, value
bl@tabl_body = remove_if	Удаляет из диапазона все объекты, удовлетворяющие predicate	first, last, predicate
remove_copy	Копирует все объекты, не равные значению value, из диапазона 1 в диапазон 2	first1, last1, first2, value
remove_copy_if	Копирует все объекты, не удовлетворяющие predicate, из диапазона 1 в диапазон 2	first1, last1, first2, pred
unique	Удаляет все эквивалентные объекты в последовательности, кроме первого	first, last
unique	Удаляет все, кроме первого, объекты в последовательности, удовлетворяющие predicate	first, last, predicate
unique_copy	Копирует только один объект из последовательности эквивалентных объектов из диапазона 1 в диапазон 2	first1, last1, first2
unique_copy	Копирует только один экземпляр из последовательности объектов, удовлетворяющих predicate, из диапазона 1 в диапазон 2	first1, last1, first2, predicate
reverse	Обращает последовательность объектов из диапазона	first, last
reverse_copy	Копирует объекты из диапазона 1 в диапазон 2 в обратном порядке	first1, last1, first2
rotate	Отражает зеркально последовательность объектов	first, last, middle

Название	Назначение	Аргументы
rotate_copy	Копирует объекты из диапазона 1 в диапазон 2, отражая их зеркально относительно итератора middle1	first1, middle1, last1, first2
random_shuffle	Перемешивает объекты в диапазоне в произвольном порядке	first, last
random_shuffle	Перемешивает объекты в диапазоне в произвольном порядке, используя функцию генерации случайного числа	first, last, rand
partition	Перемещает объекты, удовлетворяющие predicate, таким образом, чтобы они предшествовали тем, которые не удовлетворяют	first, last, predicate
stable_partition	Перемещает объекты, удовлетворяющие predicate, таким образом, чтобы они предшествовали тем, которые не удовлетворяют, при этом сохраняя относительный порядок следования в каждой из групп объектов	first, last, predicate
Операции сортировки и отношений		
sort	Сортирует объекты в диапазоне	first, last
sort	Сортирует объекты в диапазоне, используя comp в качестве функции сравнения	first, last, comp
stable_sort	Сортирует объекты в диапазоне. С поддержкой обработки порядка следования эквивалентных объектов	first, last
stable_sort	Сортирует объекты в диапазоне, используя comp в качестве функции сравнения. С поддержкой обработки порядка следования эквивалентных объектов	first , last, comp
partial_sort	Сортирует все объекты диапазона, но помещает только столько, сколько умещается между first и middle. Порядок объектов от middle до конца диапазона не определен	first, middle, last
partial_sort	Сортирует все объекты диапазона, но помещает только столько, сколько умещается между first и middle. Порядок объектов от middle до конца диапазона не определен. Для указания порядка сортировки используется predicate	first, middle, last, predicate
partial_sort_copy	То же, что предыдущий алгоритм, но помещает результирующую последовательность в диапазон 2	first1, last1, first2, last2
partial_sort_copy	То же, что partial_sort (first, middle, last, predicate), но помещает результирующую последовательность в диапазон 2	first1, last1 , first2
nth_element	Помещает n-й объект в позицию, которую он занимал бы после сортировки всего диапазона	first, nth, last
nth_element	Помещает n-й объект в позицию, которую он занимал бы после сортировки всего диапазона. Для сравнения при сортировке используется функция comp	first, nth, last, comp
lower_bound	Возвращает итератор, указывающий на первую позицию, в которую можно вставить значение value без изменения порядка следования объектов	first, last, value

продолжение ⇒

Таблица Е.1 (*продолжение*)

Название	Назначение	Аргументы
lower_bound	Возвращает итератор, указывающий на первую позицию, в которую можно вставить значение value без изменения порядка следования объектов. При определении этого порядка используется функция comp	first, last, value, comp
upper_bound	Возвращает итератор, указывающий на последнюю позицию, в которую можно вставить значение value без изменения порядка следования объектов	first, last, value
upper_bound	Возвращает итератор, указывающий на последнюю позицию, в которую можно вставить значение value без изменения порядка следования объектов. При определении этого порядка используется функция comp	first, last, value, comp
equal_range	*Возвращает пару объектов, представляющих собой* нижнюю и верхнюю границы, между которыми можно вставить значение value без изменения порядка сортировки	*first, last, value*
equal_range	Возвращает пару объектов, представляющих собой нижнюю и верхнюю границы, между которыми можно вставить значение value без изменения порядка сортировки. При сортировке используется функция comp	first, last, value, comp
binary_search	Возвращает значение «истина», если значение value входит в интервал	first, last, value
binary_search	Возвращает значение «истина», если значение value входит в интервал. Порядок сортировки определяется функцией сравнения comp	first, last, value, comp
merge	Соединяет отсортированные диапазоны 1 и 2 в диапазон 3	first1, last1, first2, last2 , first3
merge	Соединяет отсортированные диапазоны 1 и 2 в диапазон 3. Порядок сортировки определяется функцией comp	first1, last1, first2, last2 , first3, comp
inplace_merge	Соединяет два соседних упорядоченных диапазона, от first до middle и от middle до last в один диапазон first last	first, middle, last
inplace_merge	Соединяет два соседних упорядоченных диапазона, от first до middle и от middle до last в один диапазон first last, причем диапазоны упорядочены с помощью функции comp	first, middle, last, comp
includes	Возвращает значение «истина», если все объекты из диапазона first2 last2 имеются также в диапазоне first1 (только для работы с множествами и мультимножествами).	first1, last1, first2, last2
includes	Возвращает значение «истина», если все объекты из диапазона first2 last2 имеются также в диапазоне first1. Порядок сортировки определяется с помощью функции comp (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, comp

Название	Назначение	Аргументы
set_union	Создает упорядоченное объединение элементов диапазонов 1 и 2 (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3
set_union	Создает упорядоченное объединение элементов диапазонов 1 и 2, порядок сортировки определяется функцией comp (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3, comp
set_intersection	Создает упорядоченное пересечение элементов диапазонов 1 и 2, порядок сортировки определяется функцией comp (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3
set_intersection	Создает упорядоченное пересечение элементов диапазонов 1 и 2 (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3, comp
set_difference	Создает упорядоченную разность множеств, заданных диапазонами 1 и 2 (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3
set_difference	Создает упорядоченную разность множеств, заданных диапазонами 1 и 2, порядок сортировки определяется функцией comp (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3, comp
set_symmetric_difference	Создает упорядоченную симметричную разность множеств, заданных диапазонами 1 и 2 (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3
set_symmetric_difference	Создает упорядоченную симметричную разность множеств, заданных диапазонами 1 и 2, порядок сортировки определяется функцией comp (только для работы с множествами и мультимножествами).	first1, last1, first2, last2, first3, comp
push_heap	Помещает значение из last-1 в результирующую кучу (heap, область динамической памяти) диапазон от first до last	first, last
push_heap	Помещает значение из last-1 в результирующую кучу (heap) в диапазон от first до last, порядок сортировки определяется функцией comp	first, last, comp
pop_heap	Меняет значения в first и last-1. Помещает диапазон first last-1 в кучу	first, last
pop_heap	Меняет значения в first и last-1. Помещает диапазон first last-1 в кучу, упорядочивание производится с помощью функции comp	first, last, comp
make_heap	Создает кучу из значений диапазона first last	first, last
make_heap	Создает кучу из значений диапазона first last, упорядочивание с помощью функции comp	first, last, comp
sort_heap	Упорядочивает элементы в куче first last	first, last
sort_heap	Упорядочивает элементы в куче first last с помощью функции comp	first, last, comp
min	Возвращает наименьшее из a, b	a, b
min	Возвращает наименьшее из a, b в соответствии с функцией comp	a, b, comp
max	Возвращает наибольшее из a, b	a, b

продолжение

Таблица Е.1 (*продолжение*)

Название	**Назначение**	**Аргументы**
max	Возвращает наибольшее из a, b в соответствии с функцией comp	a, b, comp
max_element	Возвращает итератор, указывающий на наибольший объект в диапазоне	first, last
max_element	Возвращает итератор, указывающий на наибольший объект в диапазоне в соответствии с функцией comp	first, last, comp
min_element	Возвращает итератор, указывающий на наименьший объект в диапазоне	first, last
min_element	Возвращает итератор, указывающий на наименьший объект в диапазоне в соответствии с функцией comp	first, last, comp
lexicographical_ compare	Возвращает значение «истина», если последовательность в диапазоне 2 следует в алфавитном порядке за последовательностью диапазона 1	first1, last1, first2, last2
lexicographical_ compare	Возвращает значение «истина», если последовательность в диапазоне 2 следует в алфавитном порядке за последовательностью диапазона 1. Алфавитный порядок определяется функцией comp	first1, last1, first2, last2, comp
next_ permutation	Выполняет одну перестановку в последовательности данного диапазона	first , last
next_ permutation	Выполняет одну перестановку в последовательности данного диапазона, порядок сортировки определяется функцией comp	first , last, comp
prev_ permutation	Выполняет одну обратную перестановку в последовательности данного диапазона	first , last
prev_ permutation	Выполняет одну обратную перестановку в последовательности данного диапазона, порядок сортировки определяется функцией comp	first , last, comp
Обобщенные числовые операции		
accumulate	Последовательно применяет формулу init=init+*iter к каждому объекту диапазона	first, last, init
accumulate	Последовательно применяет формулу init=op(init,*iter) к каждому объекту диапазона	first, last, init, op
inner_product	Последовательно применяет формулу init=init+(*iter1)*(*iter2) к соответствующим объектам диапазонов 1 и 2	first1, last1, first2, init
inner_product	Последовательно применяет формулу init=op1(init, op2(*iter1,*iter2)) к соответствующим объектам диапазонов 1 и 2	first1, last1, first2, init, op1, op2
partial_sum	Складывает значения от начала диапазона до текущего итератора и помещает суммы в позиции, на которые указывают соответствующие итераторы диапазона 2. *iter2=sum(*first1, *(first1+1), *(first1+2), ..*iter1)	first1, last1, first2

Название	Назначение	Аргументы
partial_sum	Последовательно применяет op к объектам между first1 и текущим итератором и помещает суммы в позицию, на которые указывает соответствующий итератор диапазона 2 answer = *first; for(iter=first+1; iter!=iter1;iter++) op(answer,*iter); *iter2 =answer;	first1, last1, first2, op
adjacent_difference	Вычисляет разность значений соседних объектов диапазона 1 и помещает результаты в диапазон 2 *iter2=*(iter1+1)-*iter1;	first1, last1, first2
adjacent_difference	Последовательно применяет op к соседним объектам диапазона 1 и помещает результаты в диапазон 2 *iter2=op(*(iter1+1), *iter1);	first1, last1, first2, op

Методы

Для обозначения методов, выполняющих одинаковые действия над различными контейнерами, используются одинаковые имена. Однако ни один контейнерный класс не содержит абсолютно все методы. В табл. Е.2 представлено соответствие методов контейнерам. Пояснений к методам не дается: либо их имена говорят сами за себя, либо они были описаны в книге.

Сокращения в заголовке таблицы расшифровываются следующим образом:

В. — вектор

Сп. — список

Д/о — очередь с двусторонним доступом

Мн. — множество

М/м — мультимножество

От. — отображение

М/о — мультиотображение

Ст. — стек

Оч. — очередь

Пр. оч. — приоритетная очередь

Таблица Е.2. Методы

	В.	Сп.	Д/о	Мн.	М/м	От.	М/о	Ст.	Оч.	Пр. оч.
operator==	x	x	x	x	x	x	x	x	x	
operator!=	x	x	x	x	x	x	x	x	x	
operator<	x	x	x	x	x	x	x	x	x	
operator>	x	x	x	x	x	x	x	x	x	
operator<=	x	x	x	x	x	x	x	x	x	
operator>=	x	x	x	x	x	x	x	x	x	

продолжение ⇨

Таблица Е.2 (*продолжение*)

	В.	Сп.	Д/о	Мн.	М/м	От.	М/о	Ст.	Оч.	Пр. оч.
operator =	x	x	x							
operator[]	x		x			x				
operator*		x	x							
operator->		x	x							
operator()				x	x	x				
operator +			x							
operator –			x							
operator++		x	x							
operator––		x	x							
operator+=			x							
operator–=			x							
begin	x	x	x	x	x	x	x			
end	x	x	x	x	x	x	x			
rbegin	x	x	x	x	x	x	x			
rend	x	x	x	x	x	x	x			
empty	x	x	x	x	x	x	x	x	x	x
size	x	x	x	x	x	x	x	x	x	x
max_size	x	x	x	x	x	x	x			
front	x	x	x						x	
back	x	x	x						x	
push_front		x	x							
push_back	x	x	x							
pop_front		x	x							
pop_back	x	x	x							
swap	x	x	x	x	x	x	x			
insert	x	x	x	x	x	x	x			
erase	x	x	x	x	x	x	x			
find				x	x	x	x			
count				x	x	x	x			
lower_bound				x	x	x	x			
upper_bound				x	x	x	x			
equal_range				x	x	x	x			
top								x		x
push								x	x	x
pop								x	x	x
capacity	x									
reserve	x									
splice		x								
remove		x								
unique		x								

	В.	Сп.	Д/о	Мн.	М/м	От.	М/о	Ст.	Оч.	Пр. оч.
merge		x								
reverse		x								
sort		x								

Итераторы

В табл. Е.3 представлены типы итераторов, необходимые каждому из алгоритмов.

Таблица Е.3. Типы итераторов, используемых алгоритмами

	Входной	Выходной	Прямой	Двунвпр.	Случ. доступв
for_each	x				
find	x				
find_if	x				
adjacent_find	x				
count	x				
count_if	x				
mismatch	x				
equal	x				
search			x		
copy	x	x			
copy_backward	x	x			
iter_swap			x		
swap_ranges			x		
transform	x	x			
replace			x		
replace_if			x		
replace_copy	x	x			
fill			x		
fill_n		x			
generate			x		
generate_n		x			
remove			x		
remove_if			x		
remove_copy	x	x			
remove_copy_if	x	x			
unique			x		
unique_copy	x	x			
reverse				x	
reverse_copy		x			
rotate			x		

продолжение

Таблица Е.3 (*продолжение*)

	Входной	Выходной	Прямой	Двунапр.	Случ. доступа
rotate_copy		x	x		
random_shuffle					x
partition				x	
stable_partition				x	
sort					x
stable_sort					x
partial_sort					x
partial_sort_copy	x				
nth_element					x
lower_bound			x		
upper_bound			x		
equal_range			x		
binary_search			x		
merge	x	x			
inplace_merge				x	
includes	x				
set_union	x	x			
set_intersection	x	x			
set_difference	x	x			
set_symmetric_difference	x	x			
push_heap					x
pop_heap					x
make_heap					x
sort_heap					x
max_element	x				
min_element	x				
lexicographical_compare	x				
next_permutation				x	
prev_permutation				x	
accumulate	x				
inner_product	x				
partial_sum	x	x			
adjacent_difference	x	x			

Приложение Ж
Ответы и решения

Глава 1

Ответы на вопросы

1. Процедурными, объектно-ориентированными.
2. б.
3. Данные; совершают действия над ними.
4. а.
5. Сокрытием данных.
6. а, г.
7. Объекты.
8. Ложно; организационные принципы различаются.
9. Инкапсуляция.
10. г.
11. Ложно; большинство строк кода на C++ ничем не отличаются от кода на C.
12. Полиморфизмом.
13. г.
14. б.
15. б, г.

Глава 2

Ответы на вопросы

1. б, в.
2. Круглые скобки.
3. Фигурные скобки { }.

4. Это первая функция, вызываемая при запуске программы.
5. Выражением.
6. Правильный ответ:
```
//Это комментарий
/*Это тоже комментарий*/
```
7. а, г.
8. а) 4;
   б) 10;
   в) 4;
   г) 4.
9. Ложно.
10. а) целочисленная константа;
    б) символьная константа;
    в) константа в формате с плавающей запятой;
    г) имя переменное или идентификатор;
    д) имя функции.
11. Правильный ответ:
    а) `cout <<'x ';`
    б) `cout <<"Jim ";`
    в) `cout <<509;`
12. Ложно; они не равны, пока не выполнится оператор.
13. cout <<setw(10)<<george;
14. IOSTREAM
15. Правильный ответ:
```
cin >>temp;
```
16. IOMANIP
17. Строковые константы, директивы препроцессора.
18. Истинно.
19. 2.
20. Присваивание (=) и арифметические (типа + и *).
21. Правильный ответ:
```
temp +=23;
  temp =temp +23;
```
22. 1.
23. 2020.
24. Для обеспечения объявлений и других данных библиотечных функций, перегружаемых операций и объектов.
25. Библиотечных.

Решения упражнений

1. Верное решение:

```
//ex2_1.cpp
//переводит галлоны в кубические футы
#include <iostream>
using namespace std;
int main()
    {
    float gallons,cufeet;
    cout <<"\nВведите количество в галлонах:";
    cin >>gallons;
    cufeet =gallons /7.481;
  cout <<"Значение в кубических футах: " <<cufeet <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex2_2.cpp
//создает таблицу
#include <iostream>
#include <iomanip>
using namespace std;
int main()
    {
    cout <<1990 <<setw(8)<<135 <<endl
    <<1991 <<setw(8)<<7290 <<endl
    <<1992 <<setw(8)<<11300 <<endl
    <<1993 <<setw(8)<<16200 <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex2_3.cpp
//арифметическое присваивание и декремент
#include <iostream>
using namespace std;
int main()
{
int var =10;
cout <<var <<endl;  //переменная равна 10
var *=2;            //ее значение стало 20
cout <<var--<<endl; //выводит и декрементирует перем.
cout <<var <<endl;  //переменная равна 19
return 0;
}
```

Глава 3

Ответы на вопросы

1. б, в
2. Правильный ответ:

```
george !=sally
```

3. −1 — истинно; только 0 — ложно.
4. Выражение инициализации инициализирует циклическую переменную, выражение опроса переменной тестирует ее, а выражение инкремента — изменяет.
5. в, г.
6. Истинно.
7. Правильный ответ:
```
for(int j=100;j<=110;j++)
 cout <<endl <<j;
```
8. Фигурными скобками.
9. в.
10. Правильный ответ:
```
int j =100;
  while(j <=110 )
    cout <<endl <<j++;
```
11. Ложно.
12. По крайней мере, один раз.
13. Правильный ответ:
```
int j =100;
  do
    cout <<endl <<j++;
  while(j <=110 );
```
14. Правильный ответ:
```
if(age >21)
    cout <<"Yes ";
```
15. г.
16. Правильный ответ:
```
if(age >21 )
    cout <<"Yes ";
  else
    cout <<"No ";
```
17 а, в.
18. '\r'.
19. Предшествующему, заключен в скобки.
20. Переформатирования.
21. Правильный ответ:
```
switch(ch)
  {
  case 'y ':
    cout <<"Да ";
    break;
  case 'n ':
    cout <<"Нет ";
    break;
```

```
  default:
    cout <<"Неизвестный ответ ";
  }
```

22. Правильный ответ:

```
ticket =(speed >55)?1 :0;
```

23. г.
24. Правильный ответ:

```
limit ==55 &&speed >55
```

25. Унарные, арифметические, отношения, логические, условные, присваивание.
26. г.
27. В начало цикла.
28. б.

Решения упражнений

1. Верное решение:

```
//ex3_1.cpp
//выводит произведения чисел
#include <iostream>
#include <iomanip>//для setw()
using namespace std;
int main()
    {
    unsigned long n;        //число
    cout <<"\nВведите число:";
    cin >>n;                //получить число
    for(int j=1;j<=200;j++) //цикл от 1 до 200
      {
      cout <<setw(5)<<j*n <<" ";//вывести произведение
      if(j%10 ==0 )            //через каждые 10 чисел
        cout <<endl;           // начинать новую строку
      }
    return 0;
    }
```

2. Верное решение:

```
//ex3_2.cpp
//переводит фаренгейты  в градусы и обратно
#include <iostream>
using namespace std;
int main()
    {
    int response;
    double temper;
    cout <<"\nНажмите 1 для перевода из Фаренгейта в градусы Цельсия,"
    <<"\n 2 для обратного перевода:";
    cin >>response;
    if(response ==1 )
      {
```

```
    cout <<"Введите температуру (в Фаренгейтах):";
    cin >>temper;
    cout <<"В градусах Цельсия это " <<5.0/9.0*(temper-32.0);
    }
  else
    {
    cout <<"Введите температуру в гр. Цельсия:";
    cin >>temper;
    cout <<"В градусах Фаренгейта это " <<9.0/5.0*temper +32.0;
    }
  cout <<endl;
  return 0;
  }
```

3. Верное решение:

```
//ex3_3.cpp
//Создает число из отдельных цифр
#include <iostream>
using namespace std;
#include <conio.h>//для getche()
int main()
    {
    char ch;
    unsigned long total =0;      //в этой переменной – число
    cout <<"\nВведите число:";
    while((ch=getche())!='\r ' ))//выход по нажатию Enter
    total =total*10 +ch-'0';     //прибавить число к total*10
    cout <<"\nПолучилось число:" <<total <<endl;
    return 0;
    }
```

4. Верное решение:

```
//ex3_4.cpp
//Моделирует калькулятор с 4-мя функциями
#include <iostream>
using namespace std;
int main()
    {
    double n1,n2,ans;
    char oper,ch;
    do {
      cout <<"\nВведите первый операнд, операцию, второй операнд:";
      cin >>n1 >>oper >>n2;
      switch(oper)
        {
        case '+':ans =n1 +n2;break;
        case '-':ans =n1 -n2;break;
        case '*':ans =n1 *n2;break;
        case '/':ans =n1 /n2;break;
        default:ans =0;
        }
      cout <<"Ответ =" <<ans;
      cout <<"\nПродолжать (Введите 'y ' или 'n ')?";
      cin >>ch;
      } while(ch !='n' ));
    return 0;
    }
```

Глава 4

Ответы на вопросы

1. б, г.
2. Истинно.
3. Точка с запятой.
4. Правильный ответ:

```
struct time
  {
  int hrs;
  int mins;
  int secs;
  };
```

5. Ложно; только определение переменной занимает под нее место в памяти.
6. в.
7. Правильный ответ:

```
time2.hrs =11;
```

8. 18 в 16-битных системах (3 структуры на 3 целых числа на 2 байта) или 36 в 32-битных системах.
9. Правильный ответ:

```
time time1 ={11,10,59 };
```

10. Истинно.
11. Правильный ответ:

```
temp =fido.dogs.paw;
```

12. в.
13. Правильный ответ:

```
enum players {B1,B2,SS,B3,RF,CF,LF,P,C };
players joe,tom;
```

14. Правильный ответ:

```
  joe =LF;
tom =P;
```

15. а. нет.
    б. да.
    в. нет.
    г. да.
16. 0, 1, 2.
17. Правильный ответ:

```
enum speeds {obsolete=78,single=45,album=33 };
```

18. Поскольку «ложь» представляется нулем.

Решения упражнений

1. Верное решение:

```
//ex4_1.cpp
//использование структур для хранения телефонных номеров
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////////
struct phone
    {
    int area;              //код региона (3 цифры)
    int exchange:          //номер АТС (3 цифры)
    int number:            //номер абонента (4 цифры)
    };
////////////////////////////////////////////////////////////
int main()
    {
    phone ph1 ={212,767,8900 };//инициализация номера
    phone ph2;                  //определение номера
                   //получить данные от пользователя
    cout <<"\nВведите полный номер (регион, АТС, номер)";
    cout <<"\n(без начальных нулей):";
    cin >>ph2.area >>ph2.exchange >>ph2.number;
    cout <<"\nМой номер: "       //вывести номера
        <<'(' <<ph1.area <<")"
        <<ph1.exchange <<'-' <<ph1.number;
    cout <<"\nВаш номер: "
        <<'(' <<ph2.area <<")"
        <<ph2.exchange <<'-' <<ph2.number <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex4_2.cpp
//структура для моделирования точки на плоскости
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////////
struct point
    {
    int xCo;                    //координата X
    int yCo:                    //координата Y
    };
////////////////////////////////////////////////////////////
int main()
    {
    point p1,p2,p3;             //определить три точки
    cout <<"\nВведите координаты p1:";//получить две точки
    cin >>p1.xCo >>p1.yCo:      //от пользователя
    cout <<"Введите координаты p2:";
    cin >>p2.xCo >>p2.yCo:
    p3.xCo =p1.xCo +p2.xCo;     //сумма координат
    p3.yCo =p1.yCo +p2.yCo:     //p1 и p2
    cout <<"Координаты p1+p2:" //вывести сумму
        <<p3.xCo <<"," <<p3.yCo <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex4_3.cpp
//использование структуры для моделирование обьема
// помещения
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance
    {
    int feet;
    float inches;
    };
////////////////////////////////////////////////////////////////
struct Volume
    {
    Distance length;
    Distance width;
    Distance height;
    };
////////////////////////////////////////////////////////////////
int main()
    {
    float l,w,h;
    Volume room1 ={{16,3.5 },{12,6.25 },{8,1.75 }};
    l =room1.length.feet +room1.length.inches/12.0;
    w =room1.width.feet +room1.width.inches /12.0;
    h =room1.height.feet +room1.height.inches/12.0;
    cout <<"Обьем =" <<l*w*h <<" кубических футов \n ";
    return 0;
    }
```

Глава 5

Ответы на вопросы

1. г (б наполовину правильно).
2. Определении.
3. Правильный ответ:

```
void foo()
    {
    cout <<"фу :-) ";
    }
```

4. Объявлением, прототипом.
5. Тело.
6. Вызовом.
7. Описателем.
8. в.
9. Ложно.
10. Для уяснения назначения аргументов.

11. а, б, в.
12. Пустые скобки означают отсутствие аргументов.
13. Один.
14. Истинно.
15. В начале объявления и описателя.
16. void.
17. Правильный ответ:
```
main()
{
int times2(int);//прототип
int alpha =times2(37);//вызов функции
}
```
18. г.
19. Изменение исходного значения аргумента (или желание избежать копирования большого аргумента).
20. а, в.
21. Правильный ответ:
```
int bar(char);
int bar(char,char);
```
22. Быстрее, больше.
23. Правильный ответ:
```
inline float foobar(float fvar)
```
24. а, б.
25. Правильный ответ:
```
char blyth(int,float=3.14159);
```
26. Видимостью, временем жизни.
27. Функции, определенные после переменных.
28. Те, в которых она определена.
29. б, г.
30. Слева от знака равенства.

Решения упражнений

1. Верное решение:
```
//ex5_1.cpp
//функция находит площадь окружности
#include <iostream>
using namespace std;
float circarea(float radius);
int main()
   {
   double rad;
   cout <<"\nВведите радиус окружности:";
```

```
    cin >>rad;
    cout <<"Площадь равна " <<circarea(rad)<<endl;
    return 0;
    }
//---------------------------------------------------------
float circarea(float r)
    {
    const float PI =3.14159F;
    return r *r *PI;
    }
```

2. Верное решение:

```
//ex5_2.cpp
//функция возводит число в степень
#include <iostream>
using namespace std;
double power(double n,int p=2);//p has default value 2
int main()
{
double number,answer;
int pow;
char yeserno;
cout <<"\nВведите число:";//get number
cin >>number;
cout <<"Будете вводить степень (y/n)?";
cin >>yeserno;
if(yeserno =='y ' ))//пользователю нужен не квадрат //числа?
{
cout <<"Введите степень:";
cin >>pow;
answer=power(number,pow);//возвести число в степень
}
else
answer =power(number);//квадрат числа
cout <<"Ответ " <<answer <<endl;
return 0;
}
//---------------------------------------------------------
//power()
//возвращает число n, возведенное в степень p
double power(double n,int p )
{
double result =1.0;//начать с 1
for(int j=0;j<p;j++)//умножить на n
result *=n;//p раз
return result;
}
```

3. Верное решение:

```
//ex5_3.cpp
//функция устанавливает наименьшее из двух чисел в ноль
#include <iostream>
using namespace std;
int main()
    {
    void zeroSmaller(int&,int&);
    int a=4,б=7,в=11,г=9;
    zeroSmaller(a,б);
```

```
    zeroSmaller(в,г);
    cout <<"\na=" <<a <<" б==" <<б
        <<" в==" <<в <<" г==" <<г;
    return 0;
    }
//--------------------------------------------------------------
//zeroSmaller()
//устанавливает наименьшее из двух чисел в ноль
void zeroSmaller(int&first,int&second)
    {
    if(first <second )
      first =0;
    else
      second =0;
    }
```

4. Верное решение:

```
//ex5_4.cpp
//функция возвращает наибольшее из двух значений расстояний
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////////
struct Distance //Английские расстояния
    {
    int feet;
    float inches;
    };
///////////////////////////////////////////////////////////////
Distance bigengl(Distance,Distance);//объявления
void engldisp(Distance);
int main()
    {
    Distance d1,d2,d3;  //определение трех расстояний
                        //получить расстояние d1 от пользователя
    cout <<"\nВведите число футов:";cin >>d1.feet;
    cout <<"Введите число дюймов:";cin >>d1.inches;
                        //получить расстояние d2 от пользователя
    cout <<"\nВведите число футов:";cin >>d2.feet;
    cout <<"Введите число дюймов:";cin >>d2.inches;
    d3 =bigengl(d1,d2); //d3 больше, чем d1 и d2
                        //вывести все значения
    cout <<"\nd1=";engldisp(d1);
    cout <<"\nd2=";engldisp(d2);
    cout <<"\nнаибольшее - ";engldisp(d3);cout <<endl;
    return 0;
    }
//--------------------------------------------------------------
//bigengl()
//сравнивает две структуры типа Distance, возвращает
//наибольшую
Distance bigengl(Distance dd1,Distance dd2 )
    {
    if(dd1.feet >dd2.feet)     //если число футов различается,
      return dd1;              //вернуть то, которое больше
    if(dd1.feet <dd2.feet)
      return dd2;
    if(dd1.inches >dd2.inches)//если число дюймов
                              // различается,
```

```
        return dd1;             //вернуть наибольшее
      else      //дюймы или dd2, если равные значения
        return dd2;
      }
//---------------------------------------------------
//engldisp()
//выводит структуру типа Distance в футах и дюймах
void engldisp(Distance dd )
    {
    cout <<dd.feet <<"\'-" <<dd.inches <<"\"";
    }
```

Глава 6

Ответы на вопросы

1. Объявление класса описывает, как будут выглядеть объекты после их создания.
2. Класс, объекту.
3. в.
4. Правильный ответ:
```
class leverage
   {
   private:
   int crowbar;
   public:
   void pry();
   };
```
5. Ложно; и данные, и функции могут быть как скрытыми, так и об[illegible]ступными.
6. Правильный ответ:
```
leverage lever1;
```
7. г.
8. Правильный ответ:
```
lever1.pry();
```
9. `inline` (также и `private`).
`int getcrow()`.
10. Правильный ответ:
```
   {return crowbar;}
```
11. Создания (определения).
12. Класса, методом которого он является.
13. Правильный ответ:
```
leverage()
  {crowbar =0;}
```
14. Истинно.
15. а.

16. Правильный ответ:
```
int getcrow();
int leverage::getcrow()
```
17. Правильный ответ:
```
{return crowbar;}
```
18. Методы и данные по умолчанию являются скрытыми в классах, но общедоступными в структурах.
19. Три, один.
20. Вызову одного из его методов.
21. б, в, г.
22. Ложно; попытки, пусть ошибочные, бывают необходимы.
23. г.
24. Истинно.
25. Правильный ответ:
```
void aFunc(const float jerry)const;
```

Решения упражнений

1. Верное решение:
```
//ex6_1.cpp
//использование класса для демонстрации целочисленного типа
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Int //(не то же самое, что int)
    {
    private:
    int i;
    public:
    Int()                  //создание Int
      {i =0;}
    Int(int ii)            //создание и инициализация Int
      {i =ii;}
  void add(Int i2,Int i3) //складывает два значения
                           //типа Int
    {i =i2.i +i3.i;}
    void display()         //вывести Int
    {cout <<i;}
    };
////////////////////////////////////////////////////////////
int main()
    {
    Int Int1(7);           //создать и инициализировать Int
    Int Int2(11);          //создать и инициализировать Int
    Int Int3;              //создать Int
    Int3.add(Int1,Int2);  //сложение двух переменных типа Int
    cout <<"\nInt3 =";Int3.display(); //вывести результат
    cout <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex6_2.cpp
//использование класса для моделирования автоматического //турникета
#include <iostream>
using namespace std;
#include <conio.h>
const char ESC =27;//Код клавиши «Esc»
const double TOLL =0.5;//пошлина равна 50 центами
///////////////////////////////////////////////////////////
class tollBooth
   {
   private:
     unsigned int totalCars; //всего машин за день
     double totalCash;       //всего денег за день
   public:                   //конструктор
     tollBooth() : totalCars(0), totalCash(0.0)
       {}
     void payingCar()        //a car paid
       {totalCars++;totalCash +=TOLL;}
     void nopayCar()         //a car didn 't pay
       {totalCars++;}
     void display()const     //display totals
       {cout <<"\nМашины: " <<totalCars
           <<",Деньги: " <<totalCash
           <<endl;}
   };
///////////////////////////////////////////////////////////
int main()
   {
   tollBooth boothl;         //создает турникет
   char ch;

   cout <<"\nНажмите 0 для машины без оплаты,"
     <<"\n 1 для каждой оплачивающей машины."
     <<"\n Esc для выхода.\n ";
   do {
     ch =getche();           //получить символ
     if(ch =='0 ' ))         //если это 0, машина не оплачивала
       boothl.nopayCar();
     if(ch =='1' ))          //если 1, машина оплачивала
       boothl.payingCar();
     }while(ch !=ESC );      //выход из цикла по Esc
   boothl.display();         //вывод отчета
   return 0;
   }
```

3. Верное решение:

```
//ex6_3.cpp
//Использование класса для демонстрации класса «время»
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class time
   {
   private:
   int hrs,mins,secs;
   public:
   time():hrs(0),mins(0),secs(0) //конструктор без аргументов
```

```
      {}
        //конструктор с тремя аргументами
    time(int h,int m,int s):hrs(h),mins(m),secs(s)
      {}
    void display()const         //формат 11:59:59
      {cout <<hrs <<":" <<mins <<":" <<secs;}
    void add_time(time t1,time t2)//сложить две переменные
      {
      secs =t1.secs +t2.secs;     //сложить секунды
      if(secs >59 )               //если перебор,
        {secs -=60;mins++;}       //прибавить минуту
      mins +=t1.mins +t2.mins;    //сложить минуты
      if(mins >59 )          //если слишком много минут,
        {mins -=60;hrs++;}  //прибавить час
      hrs +=t1.hrs +t2.hrs; //сложить часы
      }
    };
////////////////////////////////////////////////////////////
int main()
    {
    const time time1(5,59,59);   //создание и инициализация
    const time time2(4,30,30);   //двух переменных
    time time3;//создать еще одну переменную
    time3.add_time(time1,time2); //сложить две переменные
    cout <<"time3 =";time3.display();//вывести результат
    cout <<endl;
    return 0;
    }
```

Глава 7

Ответы на вопросы

1. г.
2. Того же.
3. Правильный ответ:
   `double doubleArray [100];`
4. 0, 9.
5. Правильный ответ:
   `cout <<doubleArray [j];`
6. в.
7. Правильный ответ:
   `int coins []={1,5,10,25,50,100 };`
8. г.
9. Правильный ответ:
   `twoD [2][4]`
10. Истинно.
11. Правильный ответ:
    `float flarr [3][3]={{52,27,83},{94,73,49},{3,6,1}};`

12. Адрес в памяти.
13. а, г.
14. Массив из 1000 элементов структуры или класса employee.
15. Правильный ответ:
    ```
    emplist [16].salary
    ```
16. г.
17. Правильный ответ:
    ```
    bird manybirds [50];
    ```
18. Ложно.
19. Правильный ответ:
    ```
    manybirds [26].cheep();
    ```
20. массив, char.
21. Правильный ответ:
    ```
    char city [21](Нужен еще один байт для пустого символа.)
    ```
22. Правильный ответ:
    ```
    char dextrose []="C6H1206-H20 ";
    ```
23. Истинно.
24. г.
25. Правильный ответ:
    ```
    strcpy(blank,name).
    ```
26. Правильный ответ:
    ```
    class dog
        {
        private:
        char breed [80];
        int age;
        };
    ```
27. Ложно.
28. б, в.
29. Правильный ответ:
    ```
    int n =s1.find("cat");
    ```
30. Правильный ответ:
    ```
    s1.insert(12,"cat ");
    ```

Решения упражнений

1. Верное решение:
    ```
    //ex7_1.cpp
    //переворачивает строку
    #include <iostream>
    #include <cstring>//для strlen()
    ```

```
using namespace std;
int main()
    {
    void reversit(char [] ));  //прототип
    const int MAX =80;         //размер массива
    char str [MAX];            //строка
    cout <<"\nВведите строку:";//получить строку от
                               //пользователя
    cin.get(str,MAX);
    reversit(str);             //перевернуть строку
    cout <<"Перевернутая строка:";//и вывести ее
    cout <<str <<endl;
    return 0;
    }
//------------------------------------------------------------
//reversit()
//функция, переворачивающая строку, переданную в аргументе
void reversit(char s [] ))
    {
    int len =strlen(s);        //найти длину строки
    for(int j =0;j <len/2;j++) //поменять все
                               //символы из первой половины
      {
      char temp =s [j];        //на символы
      s [j] ==s [len-j-1];     //из второй половины
      s [len-j-1] ==temp;
      }
    }
```

2. Верное решение:

```
//ex7_2.cpp
//объект employee, использующий строковый тип данных
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class employee
    {
    private:
      string name;
      long number;
    public:
      void getdata()    //получить данные от пользователя
        {
        cout <<"\nВведите имя:";cin >>name;
        cout <<"Введите номер:";cin >>number;
        }
      void putdata()    //вывод данных
        {
        cout <<"\n Имя:" <<name;
        cout <<"\n Номер:" <<number;
        }
    };
////////////////////////////////////////////////////////////
int main()
    {
    employee emparr [100]; //массив типа employee
    int n =0;              //количество работников
```

```
    char ch;                //ответ пользователя
    do {                    //получить данные от пользователя
      cout <<"\nВведите данные о работнике с номером "
      emparr [n++].getdata();
      cout <<"Продолжить (y/n)?";cin >>ch;
      }while(ch !='n' ));
    for(int j=0;j<n;j++)    //вывести данные из массива
      {
      cout <<"\nНомер работника " <<j+1;
      emparr [j].putdata();
      }
    cout <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex7_3.cpp
//считает среднее значение длин, введенных пользователем
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance //класс  английских расстояний
    {
    private:
      int feet;
      float inches;
    public:
      Distance()                //конструктор (без аргументов)
        {feet =0;inches =0;}
      Distance(int ft,float in)//конструктор (2 аргумента)
        {feet =ft;inches =in;}
      void getdist()            //получить расстояние
                                //от пользователя
        {
        cout <<"\nВведите футы:";cin >>feet;
        cout <<"Введите дюймы:";cin >>inches;
        }
      void showdist()           //вывод расстояния
        {cout <<feet <<"\'-" <<inches <<'\"';}
      void add_dist(Distance,Distance );//объявления
      void div_dist(Distance,int );
    };
//----------------------------------------------------------
//сложение расстояний d2 и d3
void Distance::add_dist(Distance d2,Distance d3)
    {
    inches =d2.inches +d3.inches; //сложить дюймы
    feet =0;                      //(для возможного переноса)
    if(inches >=12.0)             //если сумма превышает 12.0,
      {//уменьшить число дюймов
      inches -=12.0;              //на 12.0 и
      feet++;                     //увеличить число футов
      }                           //на 1
    feet +=d2.feet +d3.feet;      //сложить футы
    }
//----------------------------------------------------------
//деление объекта Distance на целое число
void Distance::div_dist(Distance d2,int divisor)
```

```
    {
    float fltfeet =d2.feet +d2.inches/12.0://преобразовать
                                            //в формат float
    fltfeet /=divisor:                      //выполнить деление
    feet =int(fltfeet):               //получить футовую часть
    inches =(fltfeet-feet)*12.0:      //получить дюймовую часть
    }
////////////////////////////////////////////////////////////
int main()
    {
    Distance distarr [100]:       //массив из 100 обьектов типа
                                  // Distance
    Distance total(0.0.0).average:// прочие расстояния
    int count =0:          //считает количество введенных значений
    char ch:               //признак ответа пользователя
    do {
      cout <<"\nВведите расстояние ": //получить значения
      distarr [count++].getdist():    //от оператора.занести
      cout <<"\nПродолжить (y/n)?":   //их в массив
      cin >>ch:
      }while(ch !='n' )):
    for(int j=0:j<count:j++)  //сложить все расстояния
    total.add_dist(total.distarr [j] )): //в total
    average.div_dist(total.count ):      //разделить на число
    cout <<"\nСреднее:":     //вывести среднее значение
    average.showdist():
    cout <<endl:
    return 0:
    }
```

Глава 8

Ответы на вопросы

1. а, в.
2. Правильный ответ:
   ```
   x3.subtract(x2.x1):
   ```
3. Правильный ответ:
   ```
   x3 =x2 -x1:
   ```
4. Истинно.
5. Правильный ответ:
   ```
   void operator --(){count--:}
   ```
6. Нисколько.
7. б, г.
8. Правильный ответ:
   ```
   void Distance::operator ++()
       {
       ++feet:
       }
   ```

9. Правильный ответ:

```
Distance Distance::operator ++()
    {
    int f =++feet;
    float i =inches;
    return Distance(f,i);
    }
```

10. Увеличивает переменную до ее использования как незамещенный оператор ++.
11. в, д, б, а, г.
12. Истинно.
13. б, в.
14. Правильный ответ:

```
String String::operator ++()
    {
    int len =strlen(str);
    for(int j=0;j<len;j++)
    str [j] ==toupper(str [j] ))
    return String(str);
    }
```

15. г.
16. Ложно, если есть программа преобразования; в противном случае истинно.
17. б.
18. Истинно.
19. Конструктор.
20. Истинно, но человеку такое трудно будет понять.
21. г.
22. Атрибутами, операторами.
23. Ложно.
24. а.

Решения упражнений

1. Верное решение:

```
//ex8_1.cpp
//перегружаемая операция '-' находит разность расстояний
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class Distance //Класс английских расстояний
    {
    private:
      int feet;
      float inches;
    public:              //конструктор без аргументов
```

```
Distance():feet(0),inches(0.0)
    {}                  //конструктор (два аргумента)
Distance(int ft,float in):feet(ft),inches(in)
    {}
void getdist()          //получить расстояние от пользователя
    {
      cout <<"\nВведите число футов:";cin >>feet;
      cout <<"Число дюймов:";cin >>inches;
    }
void showdist()         //вывести расстояние
    {cout <<feet <<"\'-" <<inches <<'\"';}

Distance operator +(Distance );//складывает два расстояния
Distance operator -(Distance );//вычитает два расстояния
};
//---------------------------------------------------------
//добавление d2 к расстоянию
Distance Distance::operator +(Distance d2)//вернуть сумму
    {
    int f =feet +d2.feet;        //сложить футы
    float i =inches +d2.inches;  //сложить дюймы
    if(i >=12.0)        //если сумма дюймов превышает 12.0,
      {                 // уменьшить число дюймов
      i -=12.0;         // на 12.0 и
      f++;              // увеличить число футов на 1
      }       //вернуть временное значение расстояния,
    return Distance(f,i);   // равное сумме
    }
//---------------------------------------------------------
//вычитание d2 из расстояния
Distance Distance::operator-(Distance d2)//вернуть разность
    {
    int f =feet - d2.feet;              //разность футов
    float i =inches - d2.inches;        //разность дюймов
    if(i <0)          //если число дюймов меньше 0,
      {               // увеличить их количество на
      i +=12.0;       //12.0 и
      f--;            //уменьшить число футов на 1
      }               //вернуть временное расстояние,
    return Distance(f,i); // равное разности
    }
/////////////////////////////////////////////////////////
int main()
    {
    Distance dist1,dist3;   //определить расстояния
    dist1.getdist();        //получить dist1 от пользователя
    Distance dist2(3,6.25); //определить, инициализировать
                            //dist2
    dist3 =dist1 - dist2;   //разность
                            //вывести все расстояния
    cout <<"\ndist1 =";dist1.showdist();
    cout <<"\ndist2 =";dist2.showdist();
    cout <<"\ndist3 =";dist3.showdist();
    cout <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex8_2.cpp
//перегружаемая операция '+=' для конкатенации строк
#include <iostream>
#include <cstring>//для strcpy(),strlen()
using namespace std;
#include <process.h>//для exit()
///////////////////////////////////////////////////////////
class String //пользовательский строковый тип
    {
    private:
      enum {SZ =80 };      //размер объектов String
      char str [SZ];       //содержит С-строку
    public:
      String()             //конструктор без аргументов
        {strcpy(str,"");}
      String(char s [] )) //конструктор с 1 аргументом
        {strcpy(str,s);}
    void display()         //вывод строки
      {cout <<str;}
    String operator +=(String ss)//прибавление строки к
                                 // имеющейся
        {//результат сохраняется в имеющейся строке
        if(strlen(str)+strlen(ss.str)>=SZ )
          {cout <<"\nПереполнение строки";exit(1);}
        strcat(str,ss.str);      //добавить аргументную строку
        return String(str);      //вернуть временный String
        }
    };
///////////////////////////////////////////////////////////
int main()
    {
    String s1 ="С Новым Годом!"; //использует конструктор с
                                 //одним аргументом
    String s2 ="Ура, товарищи!"; //использует конструктор
                                 // с одним аргументом
    String s3;  //использует конструктор без аргументов
    s3 =s1 +=s2;// прибавить s2 к s1, результат - в s3
    cout <<"\ns1=";s1.display(); //вывести s1
    cout <<"\ns2=";s2.display(); //вывести s2
    cout <<"\ns3=";s3.display(); //вывести s3
    cout <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex8_3.cpp
//перегружаемая операция '+' складывает два времени
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////
class time
    {
    private:
      int hrs,mins,secs;
    public:
```

```
      time():hrs(0),mins(0),secs(0)//конструктор без арг.
        {}
                                    //конструктор с тремя аргументами
      time(int h,int m,int s):hrs(h),mins(m),secs(s)
        {}
    void display()                  //формат 11:59:59
      {cout <<hrs <<":" <<mins <<":" <<secs;}
    time operator + (time t2)       //сложить два времени
      {
    int s =secs +t2.secs;           //сложить секунды
    int m =mins +t2.mins;           //сложить минуты
    int h =hrs +t2.hrs;             //сложить часы
    if(s >59 )               //если слишком много секунд,
      {s -=60;m++;}          //перенести их в одну минуту
    if(m >59 )               //если слишком много минут,
      {m -=60;h++;}          //перенести их в один час
    return time(h,m,s);      //вернуть временное значение
    }
};
////////////////////////////////////////////////////////////
int main()
    {
    time time1(5,59,59);  //создать и инициализировать
    time time2(4,30,30);  //два времени
    time time3;           //еще одно время создать
    time3 =time1 +time2;  //сложить два значения времени
    cout <<"\ntime3 =";time3.display(); //вывести результат
    cout <<endl;
    return 0;
    }
```

4. **Верное решение:**

```
//ex8_4.cpp
//работа перегружаемых арифметических операций с типом Int
#include <iostream>
using namespace std;
#include <process.h>          //для exit()
////////////////////////////////////////////////////////////
class Int
    {
    private:
      int i;
    public:
      Int():i(0)              //конструктор без аргументов
        {}
      Int(int ii):i(ii)       //конструктор с одним
                              //аргументом
        {}                    //(из int в Int)
      void putInt()           //вывод Int
        {cout <<i;}
      void getInt()           //чтение Int с клавиатуры
        {cin >>i;}
      operator int()          //операция преобразования
        {return i;}           //(Int в int)
      Int operator +(Int i2) //сложение
        {return checkit(long double(i)+long double(i2));}
      Int operator -(Int i2) //вычитание
        {return checkit(long double(i)-long double(i2));}
```

```
    Int operator *(Int i2) //умножение
      {return checkit(long double(i)*long double(i2));}
    Int operator /(Int i2) //деление
      {return checkit(long double(i)/long double(i2));}
    Int checkit(long double answer) //проверка
                                    //результатов
      {
      if(answer >2147483647.0L ||answer <-2147483647.0L )
        {cout <<"\nОшибка переполнения\n ";exit(1);}
      return Int(int(answer));
      }
  };
////////////////////////////////////////////////////////////
int main()
  {
  Int alpha =20;
  Int beta =7;
  Int delta,gamma;
  gamma =alpha +beta;    //27
  cout <<"\ngamma=";gamma.putInt();
  gamma =alpha -beta;    //13
  cout <<"\ngamma=";gamma.putInt();
  gamma =alpha *beta;    //140
  cout <<"\ngamma=";gamma.putInt();
  gamma =alpha /beta;    //2
  cout <<"\ngamma=";gamma.putInt();
  delta =2147483647;
  gamma =delta +alpha;   //ошибка переполнения
  delta =-2147483647;
  gamma =delta -alpha;   //ошибка переполнения
  cout <<endl;
  return 0;
  }
```

Глава 9

Ответы на вопросы

1. а, в.
2. Порожденным.
3. б, в, г.
4. Правильный ответ:
   `class Bosworth :public Alphonso`
5. Утверждение ложно.
6. Скрытые.
7. Да (предполагая, что basefunc не скрыта).
8. Правильный ответ:
   `BosworthObj.alfunc();`
9. Истинно.

10. Тот, который принадлежит порожденному классу.
11. Правильный ответ:
    `Bosworth():Alphonso(){}`
12. в, г.
13. Истинно.
14. Правильный ответ:
    `Derv(int arg):Base(arg)`
15. а.
16. Истинно.
17. в.
18. Правильный ответ:
    `class Tire :public Wheel,public Rubber`
19. Правильный ответ:
    `Base::func();`
20. Ложно.
21. Обобщением.
22. г.
23. Ложно.
24. Сильная, агрегации.

Решения упражнений

1. Верное решение:

```
//ex9_1.cpp
//класс публикаций и порожденные из него
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class publication //базовый класс
    {
    private:
      string title;
      float price;
    public:
      void getdata()
        {
        cout <<"\nВведите заголовок:";cin >>title;
        cout <<"Введите цену:";cin >>price;
        }
      void putdata()const
        {
        cout <<"\nЗаголовок:" <<title;
        cout <<"\nЦена:" <<price;
        }
      };
```

```
/////////////////////////////////////////////////////////////
class book :private publication //порожденный класс
    {
    private:
      int pages;
      public:
        void getdata()
          {
          publication::getdata();
          cout <<"Введите число страниц:";cin >>pages;
          }
        void putdata()const
          {
          publication::putdata();
          cout <<"\nСтраниц:" <<pages;
          }
        };
/////////////////////////////////////////////////////////////
class tape :private publication //порожденный класс
    {
    private:
      float time;
    public:
      void getdata()
        {
        publication::getdata();
        cout <<"Введите время звучания:";cin >>time;
        }
      void putdata()const
        {
        publication::putdata();
        cout <<"\nВремя звучания:" <<time;
        }
      };
/////////////////////////////////////////////////////////////
int main()
    {
    book book1;                  //определить публикации
    tape tape1;
    book1.getdata();             //получить данные о них
    tape1.getdata();
    book1.putdata();             //вывести данные о них
    tape1.putdata();
    cout <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex9_2.cpp
//наследование класса String
#include <iostream>
#include <cstring>          //для strcpy().etc.
using namespace std;
/////////////////////////////////////////////////////////////
class String                //базовый класс
    {
    protected:              //Примечание: не обязательно
                            //делать их скрытыми
```

```
    enum {SZ =80 }:          //размер всех объектов String
    char str [SZ]:           //содержит С-строку
    public:
      String()               //конструктор 0, без аргументов
        {str [0] ='\0':}
      String(char s [] )) //конструктор 1, один аргумент
        {strcpy(str,s):}  //преобразование из string в
                          // String
      void display()const //вывести String
        {cout <<str:}
      operator char*()       //функция преобразования
        {return str:}        //преобразовать String в C-string
      }:
////////////////////////////////////////////////////////////
class Pstring :public String  //порожденный класс
    {
    public:
      Pstring(char s [] )):   //конструктор
    }:
//----------------------------------------------------------
Pstring::Pstring(char s [] )) //конструктор для Pstring
    {
    if(strlen(s)>SZ-1)         //если слишком длинная строка,
      {
      for(int j=0;j<SZ-1;j++) //копировать первую SZ-1
      str [j] =s [j]:          //символы «вручную»
      str [j] ='\0':           //добавить служебный символ
      }
    else                       //если не слишком длинная,
      String(s):               //создать нормальную строку
    }
////////////////////////////////////////////////////////////
int main()
    {                              //определить String
    Pstring s1 ="Это очень-очень длинная строка, которая, возможно, "
"да, нет, точно, -- превысит размер, ограниченный SZ.":
    cout <<"\ns1=":s1.display():  //вывести String
    Pstring s2 ="А это коротенькая строка."://определить
                                            //String
    cout <<"\ns2=":s2.display():  //вывести String
    cout <<endl:
    return 0:
    }
```

3. Верное решение:

```
//ex9_3.cpp
//множественное наследование класса публикаций
#include <iostream>
#include <string>
using namespace std:
////////////////////////////////////////////////////////////
class publication
    {
    private:
    string title:
    float price:
```

```
    public:
      void getdata()
        {
        cout <<"\nВведите заголовок:";cin >>title;
        cout <<"Введите цену:";cin >>price;
        }
      void putdata()const
        {
        cout <<"\nЗаголовок:" <<title;
        cout <<"\nЦена:" <<price;
        }
      };
///////////////////////////////////////////////////////////
class sales
    {
    private:
      enum {MONTHS =3 };
      float salesArr [MONTHS];
    public:
      void getdata();
      void putdata()const;
    };
//---------------------------------------------------------
void sales::getdata()
    {
    cout <<"Введите объем продаж за 3 месяца:\n ";
    for(int j=0;j<MONTHS;j++)
      {
      cout <<" Месяц " <<j+1 <<":";
      cin >>salesArr [j];
      }
    }
//---------------------------------------------------------
void sales::putdata()const
    {
    for(int j=0;j<MONTHS;j++)
      {
      cout <<"\nПродажи за месяц" <<j+1 <<":";
      cout <<salesArr [j];
      }
    }
///////////////////////////////////////////////////////////
class book :private publication,private sales
    {
    private:
      int pages;
    public:
      void getdata()
        {
        publication::getdata();
        cout <<"Введите число страниц:";cin >>pages;
        sales::getdata();
        }
      void putdata()const
        {
        publication::putdata();
```

```
        cout <<"\nСтраниц:" <<pages;
        sales::putdata();
        }
      };
////////////////////////////////////////////////////////////
class tape :private publication,private sales
    {
    private:
      float time;
    public:
      void getdata()
        {
        publication::getdata();
        cout <<"Введите время звучания:";cin >>time;
        sales::getdata();
        }
      void putdata()const
        {
        publication::putdata();
        cout <<"\nВремя звучания:" <<time;
        sales::putdata();
        }
      };
////////////////////////////////////////////////////////////
int main()
    {
    book book1;          //определить публикации
    tape tape1;
    book1.getdata();     //получить сведения о публикациях
    tape1.getdata();
    book1.putdata();     //вывести сведения о публикациях
    tape1.putdata();
    cout <<endl;
    return 0;
    }
```

Глава 10

Ответы на вопросы

1. Правильный ответ:
   `cout <<&testvar;`
2. 4 байта.
3. в.
4. Правильный ответ:
   `&var , *var , var&, char*`
5. Константа; переменная.
6. Правильный ответ:
   `float *ptrtofloat;`
7. Имени.

8. Правильный ответ:
```
*testptr
```
9. Указатель на; значение переменной, на которую ссылается указатель.
10. б, в, г.
11. Нет. Адрес &intvar должен быть помещен в указатель intptr до первого обращения к переменной.
12. Любой тип данных.
13. Оба делают одно и то же.
14. Правильный ответ:
```
for(int j=0;j<77;j++)
cout <<endl <<*(intarr+j);
```
15. Потому что имя массива — это его адрес, который нельзя изменять.
16. Ссылке; указателю.
17. а, г.
18. Правильный ответ:
```
void func(char*);
```
19. Правильный ответ:
```
for(int j=0;j<80;j++)
*s2++=*s1++;
```
20. б.
21. Правильный ответ:
```
char*revstr(char*);
```
22. Правильный ответ:
```
char*numptrs []={"Один ","Два ","Три "};
```
23. а, в.
24. Потерям.
25. Память, которая больше не используется.
26. Правильный ответ:
```
p->exclu();
```
27. Правильный ответ:
```
objarr [7].exclu();
```
28. а, в.
29. Правильный ответ:
```
float*arr [8];
```
30. б.
31. 0..9 с одного конца; 3..* — с другого.
32. б.
33. Ложно.
34. а.

Решения упражнений

1. Верное решение:

```
//ex10_1.cpp
//находит среднее чисел, введенных пользователем
#include <iostream>
using namespace std;
int main()
    {
    float flarr [100];        //массив чисел
    char ch;                  //выбор пользователя
    int num =0;               //считает количество
                              //введенных чисел
    do
      {
    cout <<"Введите число:"; //получить числа
      cin >>*(flarr+num++);  //пока не будет ответа 'n'
      cout <<"Продолжать(y/n)?";
      cin >>ch;
      }
    while(ch !='n');
      float total =0.0;       //сумма начинается с 0
      for(int k=0;k<num;k++) //прибавлять числа к сумме
      total +=*(flarr+k);
      float average =total/num;//поиск и вывод среднего
      cout <<"Среднее равно " <<average <<endl;
      return 0;
    }
```

2. Верное решение:

```
//ex10_2.cpp
//метод переводит строки в верхний регистр
#include <iostream>
#include <cstring>       //для strcpy() и т.п.
#include <cctype>        //для toupper()
using namespace std;
////////////////////////////////////////////////////////////
class String             //пользовательский строковый тип
    {
    private:
      char*str;          //указатель на строку
    public:
      String(char*s)     //конструктор с 1 аргументом
      {
      int length =strlen(s);    //длина строки из аргумента
      str =new char [length+1]; //занять память
      strcpy(str,s);            //скопировать туда строку
      }
    ~String()            //деструктор
      {delete str;}
    void display()       //вывести String
      {cout <<str;}
    void upit();         //String в верхний регистр
    };
//----------------------------------------------------------
void String::upit()      //каждый символ в верхний регистр
```

```
    {
    char*ptrch =str;      //указатель на эту строку
    while(*ptrch )        //до пустого символа,
      {
      *ptrch =toupper(*ptrch)://каждый символ в
                              //ВЕРХНИЙ РЕГИСТР
    ptrch++:              //перейти к следующему символу
    }
  }
//////////////////////////////////////////////////////////
int main()
    {
    String s1 ="Хорошо смеется тот, кто смеется последний";
    cout <<"\ns1=";       //вывести строку
    s1.display():
    s1.upit():            //в ВЕРХНИЙ РЕГИСТР
    cout <<"\ns1=";       //вывести строку
    s1.display():
    cout <<endl:
    return 0:
    }
```

3. Верное решение:

```
//ex10_3.cpp
//сортировать массив указателей на строку
#include <iostream>
#include <cstring>          //для strcmp() и т.п.
using namespace std:
const int DAYS =7:          //число указателей на массив
int main()
    {
    void bsort(char**.int): //прототип
                            //массив указателей на char
    char*arrptrs [DAYS] =={"Воскресенье ","Понедельник" ,"Вторник ","Среда ",
"Четверг ","Пятница ","Суббота " }
    }:
    cout <<"\nНеупорядоченный:\n ";
    for(int j=0:j<DAYS:j++) //вывести неупорядоченные
                            //строки
    cout <<*(arrptrs+j)<<endl:
    bsort(arrptrs.DAYS):    //сортировать строки
    cout <<"\nУпорядоченная:\n ";
    for(j=0:j<DAYS:j++)     //вывести сортированные строки
    cout <<*(arrptrs+j)<<endl:
    return 0:
    }
//--------------------------------------------------------
void bsort(char**pp.int n)  //сортировать указатели на строки
    {
    void order(char**.char**): //прототип
    int j.k:                   //индексы массива
    for(j=0:j<n-1:j++)         //внешний цикл
      for(k=j+1:k<n:k++)       //внутренний цикл
        order(pp+j.pp+k):     //упорядочить содержимое
                              // указателей
    }
```

```
//--------------------------------------------------------
void order(char**pp1,char**pp2) //сортирует два указателя
    {           //если в первом строка больше
    if(strcmp(*pp1,*pp2)>0)     //чем во втором
      {
      char*tempptr =*pp1;       //обменять указатели
      *pp1 =*pp2;
      *pp2 =tempptr;
      }
    }
```

4. Верное решение:

```
//ex10_4.cpp
//связный список включает в себя деструктор
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////
struct link             //один элемент списка
    {
    int data;           //элемент данных
    link*next;          //указатель на следующий элемент
    };
//////////////////////////////////////////////////////////
class linklist          //список ссылок
    {
    private:
      link*first;       //указатель на первый элемент
    public:
      linklist()        //конструктор без аргументов
        {first = NULL;} //первого элемента нет
      ~linklist();      //деструктор
      void additem(int r); //добавить элемент
      void display();      //вывести все ссылки
    };
//--------------------------------------------------------
void linklist::additem(int r)  //добавление элемента данных
    {
    link*newlink =new link;    //создать новую ссылку
    newlink->data = r;         //предоставить ей данные
    newlink->next = first;     //указывает на следующий
                               //элемент
    first =newlink;            //теперь первый указывает на него
    }
//--------------------------------------------------------
void linklist::display()       //вывод всех ссылок
    {
    link*current = first;      //установить указатель на
                               //первый элемент
while(current !=NULL )         //выход по достижении
                               //последнего элемента
    {
    cout <<endl <<current->data; //вывести данные
    current =current->next;      //перейти к следующей
                                 //ссылке
    }
}
//--------------------------------------------------------
linklist::~linklist()       //деструктор
```

```
    {
    link*current =first;   //установить указатель на
                           //первый элемент
    while(current !=NULL ) //выход по достижении
                           //последнего элемента
      {
      link*temp =current;  //сохранить указатель на данный
                           //элемент
        current =current->next;  //получить ссылку на
                                 // следующую ссылку
      delete temp;               //удалить эту ссылку
      }
    }
////////////////////////////////////////////////////////////
int main()
    {
    linklist li;         //создать связный список
    li.additem(25);      //добавить 4 элемента в список
    li.additem(36);
    li.additem(49);
    li.additem(64);
    li.display();        //вывести весь список
    cout <<endl;
    return 0;
    }
```

Глава 11

Ответы на вопросы

1. г.
2. Истинно.
3. Базового.
4. Правильный ответ:
   `virtual void dang(int);` или `void virtual dang(int);`
5. Поздним или динамическим связыванием.
6. Порожденного.
7. Правильный ответ:
   `virtual void aragorn()=0;`или `void virtual aragorn()=0;`
8. а, в.
9. Правильный ответ:
   `dong*parr [10];`
10. в.
11. Истинно.
12. в, г.
13. Правильный ответ:
    `friend void harry(george);`
14. а, в, г.

15. Правильный ответ
```
friend class harry;
```
или
```
friend harry;
```
16. в.
17. выполняет поэлементное копирование.
18. Правильный ответ:
```
zeta&operator =(zeta&);
```
19. а, б, г.
20. Ложно; компилятор обращается к конструктору по умолчанию.
21. а, г.
22. Правильный ответ:
```
Bertha(Bertha&);
```
23. Истинно, если была причина так делать.
24. а, в.
25. Истинно; проблема возникает, если оно возвращается по значению.
26. Они работают одинаково.
27. а, б.
28. На объект, методом которого является функция, использующая его.
29. Нет; так как this — указатель, следует использовать
```
this->da=37;
```
30. Правильный ответ:
```
return *this;
```
31. в.
32. Связями.
33. Истинно.
34. а, б, в.

Решения упражнений

1. Верное решение:
```
//ex11_1.cpp
//класс публикаций и порожденные
#include <iostream>
#include <string>
using namespace std;
///////////////////////////////////////////////////////////
class publication
    {
    private:
      string title;
      float price;
```

```
    public:
      virtual void getdata()
        {
        cout <<"\nВведите заголовок:";cin >>title;
        cout <<"Введите цену:";cin >>price;
        }
      virtual void putdata()
        {
        cout <<"\n \nЗаголовок:" <<title;
        cout <<"\nЦена:" <<price;
        }
      };
//////////////////////////////////////////////////////////
class book : public publication
    {
    private:
      int pages;
    public:
      void getdata()
        {
        publication::getdata();
        cout <<"Введите число страниц:";cin >>pages;
        }
      void putdata()
        {
        publication::putdata();
        cout <<"\nСтраниц:" <<pages;
        }
      };
//////////////////////////////////////////////////////////
class tape :public publication
    {
    private:
      float time;
    public:
      void getdata()
        {
        publication::getdata();
        cout <<"Введите время звучания:";cin >>time;
        }
      void putdata()
        {
        publication::putdata();
        cout <<"\nВремя звучания:" <<time;
        }
      };
//////////////////////////////////////////////////////////
int main()
    {
    publication*pubarr [100]; //массив указателей на
                              // публикации
    int n =0;                 //число публикаций в массиве
    char choice;              //выбор пользователя
    do {
      cout <<"\nВводить данные для книги или пленки (б/t)?";
      cin >>choice;
      if(choice=='б ' ))       //создать объект «книга»
        pubarr [n] ==new book; //занести в массив
```

```
    else                         //создать объект «пленка»
  pubarr [n] ==new tape;       //занести в массив
  pubarr [n++]->getdata();     //получить данные об объекте
  cout <<"Продолжать((y/n)?";//еще публикации?
  cin >>choice;
  }
  while(choice =='y ');        //цикл, пока не будет ответ 'y'
  for(int j=0;j<n;j++)         //цикл по всем объектам
  pubarr [j]->putdata();       //вывести данные о публикации
  cout <<endl;
  return 0;
  }
```

2. Верное решение:

```
//ex11_2.cpp
//дружественная функция square() для Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance          //класс английских расстояний
  {
  private:
    int feet;
    float inches;
  public:
    Distance()          //конструктор без аргументов
      {feet =0;inches =0.0;}
    Distance(float fltfeet) //конструктор с одним аргументом
      {                     //футы – целая часть
      feet = static_cast<int>(fltfeet);
      inches =12*(fltfeet-feet);//дюймы - в остатке
      }
                        //конструктор с двумя аргументами
    Distance(int ft,float in):feet(ft),inches(in)
      {}
    void showdist()     //вывод расстояния
      {cout <<feet <<"\'-" <<inches <<'\"';}
                        //дружественная
    friend Distance operator *(Distance,Distance);
      };
//----------------------------------------------------------
//умножить d1 на d2
Distance operator *(Distance d1,Distance d2)
  {
  float fltfeet1 =d1.feet +d1.inches/12; //преобразовать
                                         //в float
  float fltfeet2 =d2.feet +d2.inches/12;
  float multfeet =fltfeet1 *fltfeet2;    //найти результат
  return Distance(multfeet);             //вернуть временное значение
  }
////////////////////////////////////////////////////////////
int main()
  {
  Distance dist1(3,6.0);  //создать какие-то расстояния
  Distance dist2(2,3.0);
  Distance dist3;
  dist3 =dist1 *dist2;    //умножение
```

```
    dist3 =10.0 *dist3;      //умножение и преобразование
                             //вывести все расстояния
    cout <<"\ndist1 =";dist1.showdist();
    cout <<"\ndist2 =";dist2.showdist();
    cout <<"\ndist3 =";dist3.showdist();
    cout <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex11_3.cpp
//создает класс массива
//перегружает операцию присваивания и конструктор  //копирования
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Array
    {
    private:
      int*ptr;          //указатель на содержимое "array"
      int size;         //размер массива
    public:
      Array():ptr(0),size(0) //конструктор без аргументов
        {}
      Array(int s):size(s)   //конструктор
        {ptr =new int [s];}
      Array(Array&);         //конструктор копирования
      ~Array()               //деструктор
        {delete [] ptr;}
      int&operator [] ((int j)  //перегружаемая операция
                                //нижнего индекса
        {return *(ptr+j);}
      Array&operator =(Array&); //перегружаемый =operator
      };
//----------------------------------------------------------
Array::Array(Array&a)        //конструктор копирования
    {
    size =a.size;            //новый того же размера
    ptr =new int [size];     //занять место для содержимого
    for(int j=0;j<size;j++)  //копировать содержимое в новый
                             //массив
    *(ptr+j)=*(a.ptr+j);
    }
//----------------------------------------------------------
Array&Array::operator =(Array&a)//перегружаемый =operator
    {
    delete [] ptr;              //удалить старое содержимое
                                //(если было)
    size =a.size;               //создать объект того же размера
    ptr =new int [a.size];      //занять место для нового
                                //содержимого
    for(int j=0;j<a.size;j++)   //копировать содержимое в
                                // объект
    *(ptr+j)=*(a.ptr+j);
    return *this;               //вернуть этот объект
```

```
////////////////////////////////////////////////////////////
int main()
   {
   const int ASIZE =10:     //размер массива
   Array arr1(ASIZE):       //создать массив
   for(int j=0:j<ASIZE:j++) //заполнить его
                            // квадратами чисел
   arr1 [j] ==j*j:
   Array arr2(arr1):        //использовать конструктор
                            //копирования
   cout <<"\narr2:":
   for(j=0:j<ASIZE:j++)     //проверка работы с arr2
   cout <<arr2 [j] <<" ":
   Array arr3.arr4:         //создать два пустых массива
   arr4 =arr3 =arr1:        //использовать операцию
                            //присваивания
   cout <<"\narr3:":
   for(j=0:j<ASIZE:j++)     //проверка нормальной работы с
                            //arr3
   cout <<arr3 [j] <<" ":
   cout <<"\narr4:":
   for(j=0:j<ASIZE:j++)     //проверка работы с arr4
   cout <<arr4 [j] <<" ":
   cout <<endl:
   return 0:
   }
```

Глава 12

Ответы на вопросы

1. б, в.
2. ios.
3. Правильный ответ:
   `ifstream . ofstream и fstream`
4. Правильный ответ:
   `ofstream salefile ("SALES.JUN "):`
5. Истинно.
6. Правильный ответ:
   `if(foobar)`
7. г.
8. Правильный ответ:
   `fileOut.put(ch):(где ch – символ)`
9. в.
10. Правильный ответ:
    `ifile.read((char*)buff.sizeof(buff)):`
11. а, б, г.
12. Расположение байта, который будет читаться или записываться следующим.

13. Ложно; файловый указатель может быть синонимом текущей позиции.
14. Правильный ответ:
    ```
    f1.seekg(-13,ios::cur);
    ```
15. б.
16. б, в.
17. skipws приводит к тому, что символы-разделители игнорируются при вводе, поэтому cin не воспринимает окончание ввода.
18. Правильный ответ:
    ```
    int main(int argc,char *argv [])
    ```
19. PRN, LPT1.
20. Правильный ответ:
    ```
    istream&operator >>(istream&,Sample&)
    ```

Решения упражнений

1. Верное решение:

```
//ex12_1.cpp
//записывает массив
#include <iostream>
#include <fstream>          //для файловых потоков
using namespace std;
///////////////////////////////////////////////////////////
class Distance              //класс английских расстояний
      {
      private:
          int feet;
          float inches;
      public:
          Distance():feet(0),inches(0.0)//конструктор без аргументов
              {}
                            //конструктор с двумя аргументами
          Distance(int ft,float in):feet(ft),inches(in)
              {}
          void getdist()//получить расстояние от пользователя
              {
              cout <<"\nВведите футы:";cin >>feet;
              cout <<"Введите дюймы:";cin >>inches;
              }
          void showdist()               //вывести расстояние
              {cout <<feet <<"\'-" <<inches <<'\"';}
          };
///////////////////////////////////////////////////////////
int main()
      {
      char ch;
      Distance dist;      //создать обьект Distance
      fstream file;       //создать входной/выходной файл
                          //открыть его для добавления
      file.open("DIST.DAT ",ios::binary |ios::app |
ios::out |ios::in );
      do                  //данные от пользователя в файл
```

```
        {
        cout <<"\nРасстояние ":
        dist.getdist()//получить расстояние
                      //записать в файл
        file.write((char*)&dist.sizeof(dist)):
        cout <<"Продолжать (y/n)?":
        cin >>ch:
        }
    while(ch=='y '):  //выйти по 'n'
    file.seekg(0):    //установить указатель на начало
                      //файла
                      //прочитать первое значение
    file.read((char*)&dist.sizeof(dist)):
    int count =0:
    while(!file.eof())//выход по EOF
        {
        cout <<"\nРасстояние " <<++count <<":"://вывести
                                                 //расстояние
        dist.showdist():
        file.read((char*)&dist.sizeof(dist)): //считать еще
                                               //расстояние
        }
    cout <<endl:
    return 0:
    }
```

2. Верное решение:

```
//ex12_2.cpp
//имитация команды COPY
#include <fstream>              //для файловых функций
#include <iostream>
using namespace std:
#include <process.h>            //для exit()
int main(int argc.char*argv [] ))
        {
        if(argc !=3 )
            {cerr <<"\nФормат:ocopy srcfile destfile ":exit(-1):}
        char ch:                //символ для считывания
        ifstream infile:        //создать входной файл
        infile.open(argv [1] )): //открыть файл
        if(!infile )            //проверка на ошибки
            {cerr <<"\nНевозможно открытие " <<argv [1]:exit(-1):}
        ofstream outfile:       //создать выходной файл
        outfile.open(argv [2] ))://открыть его
        if(!outfile )           //проверка на ошибки
            {cerr <<"\nНевозможно открытие " <<argv [2]:exit(-1):}
        while(infile )          //до EOF
            {
            infile.get(ch):     //считать символ
            outfile.put(ch):    //записать символ
            }
        return 0:
        }
```

3. Верное решение:

```
//ex12_3.cpp
//выводит размер файла
#include <fstream>              //для файловых функций
```

```
#include <iostream>
using namespace std;
#include <process.h>            //для exit()
int main(int argc,char*argv [] ))
        {
        if(argc !=2 )
            {cerr <<"\nФормат:filename \n ";exit(-1);}
        ifstream infile;        //создать входной файл
        infile.open(argv [1] )); //и открыть его
        if(!infile )            //проверка наличия ошибок
            {cerr <<"\nНевозможно открытие" <<argv [1];exit(-1);}
        infile.seekg(0,ios::end);//перейти на конец файла
                                //сообщить номер байта
        cout <<"Размер " <<argv [1] <<" равен "
                                   <<infile.tellg();
        cout <<endl;
        return 0;
        }
```

Глава 13

Ответы на вопросы

1. а, б, в, г.
2. Директивы #include.
3. Компилятора для компилирования .CPP файла и компоновщика для связывания с .OBJ файлами.
4. а, б.
5. Библиотекой класса.
6. Истинно.
7. в, г.
8. Истинно.
9. Ложно.
10. а, в, г.
11. Связывания.
12. Ложно.
13. г.
14. Областью видимости.
15. Объектными.
16. Объявлена, файле b.
17. Истинно.
18. б.
19. Ложно.
20. г.

21. б.
22. namespace.
23. б, г.

Глава 14

Ответы на вопросы

1. б и в.
2. class.
3. Ложно; различные функции создаются в процессе компиляции.
4. Правильный ответ:
```
template<class T>
T times2(T arg)
   {
   return arg*2;
   }
```
5. б.
6. Истинно.
7. Реализацией.
8. в.
9. Фиксированный тип данных, произвольного типа данных.
10. Хранят данные.
11. в.
12. try, catch, throw.
13. Правильный ответ:
```
throw BoundsError();
```
14. Ложно; они должны быть частью блока повторных попыток.
15. г.
16. Правильный ответ:
```
class X
   {
   public:
     int xnumber;
     char xname [MAX];
     X(int xd,char*xs)
       {
       xnumber =xd;
       strcpy(xname,xs);
       }
   };
```
17. Ложно.
18. а и г.

19. г.
20. Истинно.
21. Независимый, зависимый.
22. а.
23. Ложно.
24. Дополнительную информацию.

Решения упражнений

1. Верное решение:

```
//ex14_1.cpp
//использование шаблона для нахождения среднего от значений
//элементов массива
#include <iostream>
using namespace std:
////////////////////////////////////////////////////////////
template <class atype>       //шаблон функции
atype avg(atype*array.int size)
    {
    atype total =0:
    for(int j=0:j<size:j++)  //среднее по массиву
      total +=array [j]:
      return (atype)total/size:
    }
////////////////////////////////////////////////////////////
int intArray [] =={1.3.5.9.11.13}:
long longArray [] =={1.3.5.9.11.13}:
double doubleArray [] =={1.0.3.0.5.0.9.0.11.0.13.0}:
char charArray [] =={1.3.5.9.11.13}:
int main()
    {
    cout <<"\navg(intArray)=" <<avg(intArray.6):
    cout <<"\navg(longArray)=" <<avg(longArray.6):
    cout <<"\navg(doubleArray)=" <<avg(doubleArray.6):
    cout <<"\navg(charArray)=" <<(int)avg(charArray.6)<< endl:
    return 0:
    }
```

2. Верное решение:

```
//ex14_2.cpp
//реализует очередь как шаблон
#include <iostream>
using namespace std:
const int MAX =3:
////////////////////////////////////////////////////////////
template <class Type>
class Queue
    {
    private:
      Type qu [MAX]:      //массив произвольного типа
      int head:           //индекс начала очереди (отсюда – удаление элементов)
      int tail:           //индекс хвоста очереди (сюда вставлять
                          //новые элементы)
```

```
    public:
      Queue()            //конструктор
        {head =-1;tail =-1;}
      void put(Type var) //вставка элемента в хвост
        {
        qu [++tail] ==var;
        if(tail >=MAX-1)//зациклить хвост
          tail =-1;
        }
      Type get()         //удаление элемента из начала
        {
        Type temp =qu [++head];//сохранение элемента
        if(head >=MAX-1)       //зациклить начало
          head =-1;
          return temp;         //вернуть временное
                               //значение
        }
      };
////////////////////////////////////////////////////////////
int main()
    {
    Queue<float>q1;      //q1 - обьект класса Queue<float>
    q1.put(1111.1F);     //занести 3
    q1.put(2222.2F);
    q1.put(3333.3F);
    cout <<"1:" <<q1.get()<<endl; //получить 2
    cout <<"2:" <<q1.get()<<endl;
    q1.put(4444.4F);              //занести 2
    q1.put(5555.5F);
    cout <<"3:" <<q1.get()<<endl; //получить 1
    q1.put(6666.6F);              //занести 1
    cout <<"4:" <<q1.get()<<endl; //получить 3
    cout <<"5:" <<q1.get()<<endl;
    cout <<"6:" <<q1.get()<<endl;
    Queue<long>q2;     //q2 - обьект класса Queue<long>
    q2.put(123123123L);//занести 3 (long), получить 2
    q2.put(234234234L);
    q2.put(345345345L);
    cout <<"1:" <<q2.get()<<endl;
    cout <<"2:" <<q2.get()<<endl;
    cout <<"3:" <<q2.get()<<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex14_3.cpp
//реализует очередь в виде шаблона
//использует механизм исключений для обработки ошибок
#include <iostream>
using namespace std;
const int MAX =3;
////////////////////////////////////////////////////////////
template <class Type>
class Queue
    {
    private:
      Type qu [MAX];       //массив произвольного типа
      int head;            //индекс начала массива (т. е.,
                           //очереди. Отсюда будут удаляться
```

```
                        //старые элементы)
     int tail:          //индекс хвоста очереди, куда будут
                        //добавляться новые элементы
int count:              //число элементов в очереди
public:
class full {}:          //классы исключений
class empty {}:
//--------------------------------------------------------
Queue()                 //конструктор
   {head =-1:tail =-1:count =0:}
void put(Type var)      //добавление элемента в конец
   {
   if(count >=MAX)      //если очередь заполнена,
     throw full():      //выдать исключение
   qu [++tail] ==var:   //сохранить элемент
   ++count:
   if(tail >=MAX-1)     //зациклить хвост
     tail =-1:
   }
//--------------------------------------------------------
Type get()              //удаление элемента из начала
   {
   if(count <=0)        //если очередь пуста,
     throw empty():     //выдать исключение
   Type temp =qu [++head]: //получить элемент
   --count:
   if(head >=MAX-1)        //зациклить начало
   head =-1:
   return temp:            //вернуть элемент
   }
}:
//////////////////////////////////////////////////////////
int main()
   {
   Queue<float>q1:    //q1 - объект класса Queue<float>
   float data:        //элемент данных, получаемый от пользователя
   char choice ='p': //'x '.'p ' или 'g '
   do                 //цикл (введите 'x' для выхода)
     {
     try              //блок повторных попыток
       {
cout <<"\nВведите 'x' для выхода,'p' для добавления,'g' для выдачи:":
       cin >>choice:
       if(choice=='p ')
         {
         cout <<"Введите значение:":
         cin >>data:
         q1.put(data):
         }
       if(choice=='g ')
         cout <<"Data=" <<q1.get()<<endl:
       }              //конец блока повторных попыток
   catch(Queue<float>::full)
     {
     cout <<"Ошибка: очередь заполнена." <<endl:
     }
   catch(Queue<float>::empty)
     {
     cout <<"Ошибка: очередь пуста." <<endl:
```

```
    }
  }while(choice !='x');
  return 0;
  }                    //конец main()
```

Глава 15

Ответы на вопросы

1. а, б, г.
2. Вектор, список, очередь с двусторонним доступом.
3. Множество, отображение.
4. а.
5. Истинно.
6. в.
7. Ложно.
8. Итераторов.
9. Функциональный объект.
10. в.
11. Ложно; он просто возвращает значение.
12. 3, 11.
13. Дублирующие.
14. б, в.
15. Указывается на.
16. Ложно.
17. Двунаправленный.
18. *iter++
19. г.
20. в.
21. Истинно.
22. Итераторов.
23. Это строка, используемая для разделения выводимых значений.
24. б.
25. Упорядочивания элементов.
26. Истинно.
27. Пары (или ассоциации).
28. Ложно.
29. а, г.
30. Конструкторов.

Решения упражнений

1. Верное решение:

```
//ex15_1.cpp
//тип float хранится в массиве, сортируется функцией sort()
#include <iostream>
#include <algorithm>
using namespace std;
int main()
    {
    int j=0,k;
    char ch;
    float fpn,farr [100];
    do {
      cout <<"Введите число в формате с плавающей запятой:";
      cin >>fpn;
      farr [j++] =fpn;
      cout <<"Продолжать('y ' или 'n '):";
      cin >>ch;
      }while(ch =='y ');
    sort(farr,farr+j);
    for(k=0;k<j;k++)
      cout <<farr [k] <<", ";
    cout <<endl;
    return 0;
    }
```

2. Верное решение:

```
//ex15_2.cpp
//использование вектора для работы со
//строками,push_back() и []
#include <iostream>
#include <string>
#pragma warning (disable:4786) //для Microsoft
#include <vector>
#include <algorithm>
using namespace std;
int main()
    {
    vector<string>vectStrings;
    string word;
    char ch;
    do {
      cout <<"Введите слово:";
      cin >>word;
      vectStrings.push_back(word);
      cout <<"Продолжать ('y ' или 'n '):";
      cin >>ch;
      }while(ch =='y ');
    sort(vectStrings.begin(),vectStrings.end());
    for(int k=0;k<vectStrings.size();k++)
      cout <<vectStrings [k] <<endl;
    return 0;
    }
```

3. Верное решение:

```
//ex15_3.cpp
//самодельный алгоритм reverse() переворачивает список
```

```
#include <iostream>
#include <list>
using namespace std;
int main()
    {
    int j;
    list<int>theList;
    list<int>::iterator iter1;
    list<int>::iterator iter2;
    for(j=2;j<16;j+=2)            //заполнить список: 2,4,6....
    theList.push_back(j);
    cout <<"До переворачивания:"; //вывести список
    for(iter1=theList.begin();iter1 !=theList.end();iter1++)
      cout <<*iter1 <<" ";
      iter1=theList.begin();      //установить на первый эл-т
      iter2 =theList.end();       //установить после последнего
      --iter2;                    //перейти на последний
      while(iter1 !=iter2)
        {
        swap(*iter1,*iter2);      //поменять начало и конец
        ++iter1;          //сдвинуться вперед от начала
        if(iter1==iter2)  //если четное число элементов
          break;
        --iter2;          //сдвинуться назад от конца
        }
      cout <<"\nПосле переворачивания:"; //вывести список
      for(iter1=theList.begin();iter1 !=theList.end();iter1++)
        cout <<*iter1 <<" ";
      cout <<endl;
      return 0;
      }
```

4. Верное решение:

```
//ex15_4.cpp
//мультимножество автоматически сортирует персон,
//хранящихся по указателям
#include <iostream>
#include <set>
#pragma warning (disable:4786)
#include <string>
using namespace std;
class person
    {
    private:
      string lastName;
      string firstName;
      long phoneNumber;
    public:
      person():            //конструктор по умолчанию
        lastName("пусто"),firstName("пусто"), phoneNumber(0L)
        { }
                           //конструктор с тремя аргументами
      person(string lana,string fina,long pho):
        lastName(lana),firstName(fina),phoneNumber(pho)
        { }
      friend bool operator<(const person&,const person&);
      void display()const   //display person 's data
        {
        cout <<endl <<lastName <<",\t " <<firstName
```

```
          <<"\t \tPhone:" <<phoneNumber;
        }
      long get_phone()const //return phone number
        {return phoneNumber;}
};                    //конец класса person
//---------------------------------------------------------
//перегружаемая операция  < для класса person
bool operator<(const person&p1,const person&p2)
    {
    if(p1.lastName ==p2.lastName)
      return (p1.firstName <p2.firstName)?true :false;
    return (p1.lastName <p2.lastName)?true :false;
    }
//---------------------------------------------------------
//функциональный объект для сравнения person с
//использованием указателей
class comparePersons
    {
    public:
      bool operator()(const person*ptrP1,
      const person*ptrP2)const
        {return *ptrP1 <*ptrP2;}
    };
/////////////////////////////////////////////////////////////
int main()
    {    //мультимножество указателей на объекты person
    multiset<person*,comparePersons>setPtrsPers;
    multiset<person*,comparePersons>::iterator iter;
                                //создать персон
  person*ptrP1 =new person("KuangThu ","Bruce ",4157300);
  person*ptrP2 =new person("McDonald ","Stacey ",3327563);
  person*ptrP3 =new person("Deauville ","William ",8435150);
  person*ptrP4 =new person("Wellington ","John ",9207404);
  person*ptrP5 =new person("Bartoski ","Peter ",6946473);
  person*ptrP6 =new person("McDonald ","Amanda ",8435150);
  person*ptrP7 =new person("Fredericks ","Roger ",7049982);
  person*ptrP8 =new person("McDonald ","Stacey ",7764987);

    setPtrsPers.insert(ptrP1);//занести данные в мультимножество
    setPtrsPers.insert(ptrP2);
    setPtrsPers.insert(ptrP3);
    setPtrsPers.insert(ptrP4);
    setPtrsPers.insert(ptrP5);
    setPtrsPers.insert(ptrP6);
    setPtrsPers.insert(ptrP7);
    setPtrsPers.insert(ptrP8);
                                  //вывести мультимножество
    cout <<"\n \nМножество упорядочено при создании:";
    for(iter=setPtrsPers.begin(); iter != setPtrsPers.end(); iter++)
    (**iter).display();
    iter =setPtrsPers.begin();//удалить всех
    while(iter !=setPtrsPers.end())
      {
      delete *iter;             //удалить сведения о персоне
      setPtrsPers.erase(iter++); //и указатель
      }
    cout <<endl;
    return 0;
    }//КОНЕЦ main()
```

Глава 16

Ответы на вопросы

1. Ложно.
2. в, г.
3. Действие.
4. Истинно.
5. Колонках.
6. а, в.
7. Ассоциации, обобщения, агрегации.
8. а, г.
9. Ложно.
10. а.
11. Истинно.
12. Ложно.
13. а, б, в, г.
14. Люди, другие системы; программами.
15. б.
16. б, в.
17. Ложно.
18. а, г.
19. б, г.
20. Ложно.
21. в, г.
22. Объектов.
23. Истинно.
24. а, г.

Приложение 3
Библиография

- Углубленное изучение C++
- Определяющие документы
- UML
- История C++
- Другое

В этом приложении вы найдете информацию о некоторых книгах, которые могут представлять интерес для студентов, изучающих C++.

Углубленное изучение C++

После того как вы стали хорошо разбираться в основах языка, следует продвигаться дальше. Мы советуем следующие книги: *Effective C++*, Scott Meyers (Addison Wesley, 1997) и *More Effective C++*, автором книги также является Scott Meyers (Addison Wesley, 1996). В этих книгах вы найдете, соответственно, «50 способов улучшить стиль программирования и дизайна программ» и «35 новых способов улучшить стиль программирования и дизайна программ». Все темы представлены довольно компактно, но изложены ясно и четко. Эти книги содержат в себе несомненно важные для программирования идеи, они очень популярны среди тех, кто пишет на C++.

Thinking in C++, Bruce Eckel (Prentice Hall, 1995) – книга, пожалуй, для начинающих, желающих быстро освоить язык; в ней можно найти хорошее описание основ C++ и прекрасные объяснения принципов его работы.

C++ FAQs (Frequently Asked Questions), Marshall Cline и Greg Lomow (Addison Wesley, 1995). В этой книге содержатся сотни тем, касающихся C++, представленных в краткой форме вопросов и ответов. Читается очень легко и, кроме того, внесет значительный вклад в ваше понимание этого языка.

C++ Distilled, Ira Pohl (Addison Wesley, 1997). Это краткий обзор основных характерных черт C++. Прекрасный справочник, который может помочь, если вы вдруг забыли синтаксис какого-то оператора и хотите срочно его найти.

Основополагающие документы

Поскольку автор языка — это человек, который его создал и знает о нем все, то основной книгой о C++ является *The C++ Programming Language, третье издание*, написанное Bjarne Stroustrup (Addison Wesley, 1997). У любого серьезного программиста, пишущего на C++, вы найдете экземпляр этого издания. Книга предполагает определенный уровень подготовки, она совсем не для начинающих.

Тем не менее все темы изложены очень ясно, и если вы уже владеете основами, то книга — для вас. На самом деле данное издание вам не особо поможет до тех пор, пока вы не изучите C++ довольно тщательно.

The Final Draft Information Standard (FDIS) for the C++ Programming Language, X3J16/97-14882 можно найти в научном центре под названием Information Technology Council (NSTIC), Washington, D. C.

Раньше определяющим документом C++ был *The Annotated C++ Reference Manual*, Margaret Ellis and Bjarne Stroustrup (Addison Wesley, 1990). Вот это действительно было нечто непростое и полное загадочных пояснений. К счастью, документ уже устарел.

UML

Издательство Addison Wesley выпускает абсолютно все (хорошие) книги, касающиеся UML. По крайней мере, ниже представлена исключительно их продукция.

Первые две книги написаны Grady Booch, James Rumbaugh и Ivar Jacobson, это, собственно говоря, авторы UML, и, надо думать, они знают, о чем говорят.

The Unified Modeling Language User Guide (1998) — вот что написано на обложке первой книги. В ней UML описывается очень детально, затрагиваются как основы, так и глубины языка, причем делается это довольно доступным языком, что делает книгу полезной даже для начинающих.

Основная часть *The Unified Modeling Language Reference Manual* (1998) состоит из алфавитного списка конструкций и терминов UML. Если вы хоть что-нибудь знаете об этом языке технологов программирования, то это издание станет для вас удобным справочником.

UML Distilled, Second Edition (1999), Martin Fowler и Kendall Scott. Это, по сути дела, самоучитель по UML. Он написан более доступным языком, чем два предыдущих документа, но не такой всеохватывающий. Хорошее подспорье для начинающего.

Книга *Using UML*, Perdita Stevens и Rob Pooley (2000), была написана в качестве университетского учебника по UML. Она читается легко и затрагивает основные темы, нужные для практической работы.

Advanced Use Case Modeling (2001), Frank Armour и Granville Miller. Здесь рассказывается обо всем, что вы хотели знать, но боялись спросить о прецедентах использования в процессах разработки программного обеспечения.

UML in a Nutshell (1998), Sinan Si Alhir. Полезный справочник. Это не учебник и не самоучитель, поэтому нужно хоть что-нибудь понимать в UML, чтобы пользоваться этой книгой.

История C++

The Design and Evolution of C++, Bjarne Stroustrup (Addison Wesley, 1994). Автор C++ описывает историю его развития. Это интересно и по-своему полезно, помогает полюбить, оценить и даже в какой-то мере понять C++.

Ruminations on C++, Andrew Koenig (Addison Wesley, 1997). Это размышления о языке в довольно неформальной форме, написанные одним из пионеров C++. Читается довольно легко и позволяет посмотреть свежим взглядом на программирование.

И другое

C++ IOStreams Handbook, Steve Teale (Addison Wesley, 1993). Приводится детальное описание потоков и файлов C++. Здесь содержится материал, который вы больше нигде не найдете!

The Standard Template Library, Alexander Stepanov и Meng Lee (Hewlett-Packard. 1994). Это основной документ, касающийся STL. Из него можно много чего узнать теоретически, но примеров, конечно, приводится мало. Есть книги, которые читаются проще, например *STL Tutorial and Reference Guide, второе издание*, David R. Musser, Gillmer J. Derge, и Atul Saini (Addison Wesley, 2001).

Object-Oriented Design in Java, Stephen Gilbert и Bill McCarty (Waite Group Press, 1998). Несмотря на то что речь идет о языке Java, это простая и понятная книга для тех, кто хочет узнать основные принципы ООП.

Windows Game Programming for Dummies, Andre LaMothe (IDG Books, 1998). Замечательная книга о подробностях программирования компьютерных игр. В ней, кроме всего прочего, описано, как использовать подпрограммы для работы с консольной графикой в Windows, которые были взяты за основу при написании библиотеки упрощенной консольной графики, см. приложение Д. Если вы интересуетесь написанием игр, обязательно купите книгу Andre LaMothe.

The C Programming Language, второе издание, Brian Kernighan и Dennis Ritchie (Prentice Hall PTR, 1988). Это основная книга по C, языку, на основе которого был построен C++. Книга не для «чайников», но если вы уже что-то знаете про C, то она будет хорошим справочным пособием.

Алфавитный указатель

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

В

Г

Д

З

И

К

Л

М

П

Т

У

Ф

Р. ЛАФОРЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ в C++

4-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Изучение объектно-ориентированной парадигмы программирования обычно ассоциируется с многочасовым курсом лекций и утомительными лабораторными занятиями, во время которых вас бросает из крайности в крайность — от тупого отчаяния до ложного ощущения полного понимания. Книга популярного среди американских студентов писателя Роберта Лафоре призвана помочь изучающему C++ избежать таких нагрузок на психику и без чьей-либо помощи пройти путь от застенчивого ученика до уверенного в своих силах программиста.

Столь значительный объем этого учебника обусловлен как сложностью предмета, так и стремлением автора не оставить белых пятен в программистском образовании читателя: любой элемент теории немедленно иллюстрируется небольшим примером кода, для приобретения практических навыков даются детально прокомментированные тексты полномасштабных приложений, а для закрепления полученных знаний читателю предлагается выполнить множество самостоятельных упражнений.

ISBN 5-94723-302-9





SAMS